MCS-48 AND UPI-41 ASSEMBLY LANGUAGE MANUAL

Manual Order Number: 9800255C

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by Intel.

The following are trademarks of Intel Corporation and may be used only to describe Intel products:

ICE
INSITE
INTEL
INTELLEC
LIBRARY MANAGER
MCS
MEGACHASSIS
MICROMAP
PROMPT
RMX
UPI
µSCOPE

PREFACE

Intel welcomes you as a new user of the Microcomputer-System/48 (MCS-48) and Universal-Peripheral-Interface/41 (UPI-41) microcomputer systems. This manual is one of a series of documents describing these systems and their operation.

Part One of this manual describes the assembly language for programming the families of MCS-48 and UPI-41 single-device microcomputers. Additional information needed to create a source (input) file to the 8048/8041 assemblers, specifically the use of assembler directives, is also included in this part of the manual.

Part Two describes procedures and controls for operating the assemblers used to translate your source file into object code recognized by the MCS-48 and UPI-41 microcomputer systems. Paper-tape-resident and diskette-resident versions of the assembler are available on Intel's Intellec Microcomputer Development System. The Intellec Series II Microcomputer Development System provides a ROM-resident assembler on the Model 210 and a diskette-resident version on the Models 220 and 230.
If you are using the ROM-resident assembler, you will need the document:

Intellec Series II Model 210 User's Guide	9800557

This manual provides only an overview of MCS-48 and UPI-41 hardware and assumes you are familiar with the documents:

MCS-48 User's Manual	9800270
UPI-41 User's Manual	9800504

If you are not already conversant with the Intellec System and its operation, please refer to the document:

MDS-800 Intellec Microcomputer Development System Operator's Manual	9800129

If you are using the diskette-resident version of the assembler (ASM48), you will need:

ISIS-II System User's Guide	9800306

Finally, you may find the following application notes useful in designing programs for the MCS-48 and UPI-41 microcomputer systems.

Application Techniques for The MCS-48 Family	AP-24
Printer Control Using The UPI-41	AP-27

FUNCTIONAL OVERVIEW 1

ASSEMBLER CONCEPTS 2

MCS-48 ASSEMBLY LANGUAGE INSTRUCTIONS 3

UPI-41 ASSEMBLY LANGUAGE INSTRUCTIONS 4

ASSEMBLER DIRECTIVES 5

MACROS 6

ASSEMBLER OVERVIEW 7

ASSEMBLER CONTROLS 8

ASSEMBLER OPERATION 9

MCS-48 AND UPI-41 INSTRUCTION SUMMARY A

ASSEMBLER DIRECTIVE SUMMARY B

ASSEMBLER CONTROL SUMMARY C

LIST FILE FORMATS D

REFERENCE TABLES E

ERROR MESSAGES F

CONTENTS

PART ONE: PROGRAMMING THE MCS-48 AND UPI-41 MICROCOMPUTER FAMILIES

1. Functional Overview

8048 Basic Features 1-1

Program Memory 1-2
Data Memory 1-3
Addressing Data Memory 1-4
Working Registers 1-5
Program Counter Stack 1-5
Programmable Controls 1-6
Carry Bit 1-7
Auxiliary Carry Bit 1-7
Flag Bits (F0,F1) 1-7
Register Bank Switch 1-8
Test Input 0 1-8
Test Input 1 1-8
Timer Flag 1-8
Interrupt Input Pin 1-8
Program Status Word 1-9
Interrupts 1-9
Input/Output 1-10

UPI-41 Microcomputers 1-11

Functional Differences 1-11
Hardware Differences 1-12
Data Bus Buffer 1-12
Status Register 1-12
8021 Microcomputer 1-13
Functional Differences 1-13
Hardware Differences 1-13

2. Assembler Concepts

Assemblers and Assembly Language 2-1

Instruction Format 2-1

Label Field 2-2
Opcode Field 2-2
Operand Field 2-2
Comment Field 2-3

Arithmetic Operations 2-3

Number Base Representation 2-4
Permissible Range of Numbers 2-4
Two's Complement Arithmetic 2-5
Assembly-Time Expression Evaluation 2-5
Operators 2-6
Precedence of Operators 2-9

Symbols and Symbol Tables 2-9

Symbolic Addressing 2-9
Symbol Characteristics 2-10
Reserved, User-Defined, and Assembler-Generated Symbols 2-10
Global and Limited Symbols 2-12
Permanent and Redefinable Symbols 2-12
Duplicate Symbols 2-12

3. MCS-48 Assembly Language Instructions

Data Transfer Instructions 3-2

Data Transfer Within 8048 Memory 3-2
Register/Accumulator Moves 3-2
Move Register Contents to Accumulator 3-2
Move Accumulator Contents to Register 3-2
Data-Memory/Accumulator Moves 3-3
Move Data Memory Contents to Accumulator 3-3
Move Accumulator Contents to Data Memory 3-3
Move External-Data-Memory Contents to Accumulator 3-4
Move Accumulator Contents to External Data Memory 3-4
Immediate-Data Moves 3-4
Move Immediate Data to Register 3-5
Move Immediate Data to Data Memory 3-5
Move Immediate Data to Accumulator 3-5
PSW/Accumulator Moves 3-6
Move PSW Contents to Accumulator 3-6
Move Accumulator Contents to PSW 3-6
Timer/Accumulator Moves 3-7
Move Timer/Counter Contents to Accumulator 3-7
Move Accumulator Contents to Timer/Counter 3-7

Program-Memory/Accumulator Moves 3-7

Move Current Page Data to Accumulator 3-8

Move Page 3 Data to Accumulator 3-8

Data Exchange Instructions 3-9

Exchange Accumulator-Register Contents 3-9

Exchange Accumulator and Data Memory Contents 3-9

Exchange Accumulator and Data Memory 4-Bit Data 3-10

Swap 4-Bit Data Within Accumulator 3-10

Input/Output Data Transfers 3-11

Standard I/O Transfers 3-11

Input Port 0-2 Data to Accumulator 3-11

Strobed Input of BUS Data to Accumulator 3-12

Output Accumulator Data to Port 0-2 3-12

Output Accumulator Data to BUS 3-12

Expanded I/O Transfers 3-13

Move Port 4-7 Data to Accumulator 3-13

Move Accumulator Data to Port 4, 5, 6, or 7 3-13

Data Manipulation Instructions 3-14

Logical Operations 3-14

Accumulator/Register Logical Operations 3-14

Logical AND Accumulator With Register Mask 3-14

Logical OR Accumulator With Register Mask 3-15

Logical XOR Accumulator With Register Mask 3-15

Accumulator/Data-Memory Logical Operations 3-15

Logical AND Accumulator With Memory Mask 3-16

Logical OR Accumulator With Memory Mask 3-16

Logical XOR Accumulator With Memory Mask 3-16

Accumulator/Immediate-Data Logical Operations 3-17

Logical AND Accumulator Wth Immediate Mask 3-17

Logical OR Accumulator With Immediate Mask 3-17

Logical XOR Accumulator With Immediate Mask 3-18

Input/Output Logical Operations 3-18

Logical AND Port 1-2 With Immediate Mask 3-18

Logical AND BUS With Immediate Mask 3-18

Logical OR Port 1-2 With Immediate Mask 3-19

Logical OR BUS With Immediate Mask 3-19

Logical AND Port 4-7 With Accumulator Mask 3-19

Logical OR Port 4-7 With Accumulator Mask 3-20

Rotate Operations 3-20

Rotate Without Carry 3-21

Rotate Right Without Carry 3-21

Rotate Left Without Carry 3-21

Rotate Through Carry 3-22

Rotate Right Through Carry 3-22

Rotate Left Through Carry 3-22

Arithmetic Operations 3-22
Increment/Decrement Instructions 3-23
Increment Accumulator 3-23
Increment Register 3-23
Increment Data Memory Location 3-24
Decrement Accumulator 3-24
Decrement Register 3-24
Decrement Register and Test 3-25
Addition Instructions 3-25
Add Register Contents to Accumulator 3-26
Add Carry and Register Contents to Accumulator 3-26
Add Data Memory Contents to Accumulator 3-27
Add Carry and Data Memory Contents to Accumulator 3-27
Add Immediate Data to Accumulator 3-27
Add Carry and Immediate Data to Accumulator 3-28
Miscellaneous Accumulator Operations 3-28
Clear Accumulator 3-28
Complement Accumulator 3-28
Decimal Adjust Accumulator 3-29

Setting Program Controls 3-29

Carry and Flag Controls 3-29
Clear Carry Bit 3-29
Complement Carry Bit 3-30
Clear Flag 0 3-30
Complement Flag 0 3-30
Clear Flag 1 3-30
Complement Flag 1 3-31
Interrupt Controls 3-31
External Interrupt Control 3-31
Enable External Interrupt 3-31
Disable External Interrupt 3-31
Timer/Counter Interrupt Control 3-32
Enable Timer/Counter Interrupt 3-32
Disable Timer/Counter Interrupt 3-32
Timer/Event-Counter Controls 3-32
Start Timer 3-32
Start Event Counter 3-33
Stop Timer/Event-Counter 3-33
Clock Control 3-34
Enable Clock Output 3-34
Memory and Register Bank Controls 3-35
Memory Bank Selection 3-35
Select Memory Bank 0 3-35
Select Memory Bank 1 3-36
Register Bank Selection 3-36
Select Register Bank 0 3-36
Select Register Bank 1 3-36

The 'Null' Operation 3-37

The NOP Instruction 3-37

Transferring Program Control 3-37

Subroutine Call/Return Operations 3-38

Subroutine Call 3-38

Return Without PSW Restore 3-39

Return With PSW Restore 3-39

Jump Instructions 3-40

Unconditional Jumps 3-40

Direct Jump Within 2K Block 3-40

Indirect Jump Within Page 3-40

Conditional Jumps 3-40

Jump If Carry Is Set 3-41

Jump If Carry Is Not Set 3-41

Jump If Accumulator Is Zero 3-41

Jump If Accumulator Is Not Zero 3-42

Jump If Flag 0 Is Set 3-42

Jump If Flag 1 Is Set 3-42

Jump If Test 0 Is High 3-43

Jump If Test 0 Is Low 3-43

Jump If Test 1 Is High 3-43

Jump If Test 1 Is Low 3-44

Jump If Timer Flag Is Set 3-44

Jump If Interrupt Input Is Low 3-44

Jump If Accumulator Bit Is Set 3-45

Sample Programs 3-45

Addition With 8-Bit Quantities 3-45

Addition With 16-Bit Quantities 3-45

Addition With BCD Quantities 3-45

Subtraction With 8-Bit Quantities 3-46

Subtraction With 16-Bit Quantities 3-46

Multiplication (8 X 8 Bits, 16-Bit Product) 3-47

Compare Memory to Accumulator 3-47

Analog/Digital Conversion 3-47

Matrix Keyboard Scan (4 x 4) 3-48

4. UPI-41 Assembly Language Instructions

Deleted 8048 Instructions 4-1

Reinterpreted Instructions 4-1

Added Instructions 4-2

Data Transfer Instructions 4-2

Input DBB Contents to Accumulator 4-2

Output Accumulator Contents to DBB 4-2

Flag Test Instructions 4-2

Jump If IBF Is Not Set 4-3

Jump If OBF Is Set 4-3

5. Assembler Directives

Location Counter Control 5-2

ORG Directive 5-2

Symbol Definition 5-2

EQU Directive 5-3

SET Directive 5-3

Data Definition 5-3

DB Directive 5-3

DW Directive 5-4

Memory Reservation 5-4

DS Directive 5-4

Conditional Assembly 5-5

IF, ELSE, and ENDIF Directives 5-5

Assembler Termination 5-7

END Directive 5-7

End-of-Tape Indication 5-7

EOT Directive 5-7

6. Macros

Introduction to Macros 6-1

Why Use Macros? 6-1

What is a Macro? 6-1

Macros Vs. Subroutines 6-2

Using Macros 6-3

Macro Definition 6-3
Macro Definition Directives 6-4
MACRO Directive 6-4
ENDM Directive 6-5
LOCAL Directive 6-5
REPT Directive 6-6
IRP Directive 6-6
IRPC Directive 6-7
EXITM Directive 6-8
Null Macros 6-8
Special Operators 6-9
Nested Macro Definitions 6-10
Macro Calls 6-11
Macro Call Format 6-11
Nested Macro Calls 6-12
Macro Expansion 6-13

Sample Macros 6-14

Repetitive Addition (IRP) 6-14
Repetitive Add and Store (IRPC, &) 6-15
Multiplication (REPT, LOCAL) 6-15
Zero and Label Contiguous Locations (REPT, &, %) 6-16
Altering Macro Expansions (Three Approaches) 6-17
First Solution (Conditional Assembly) 6-17
Second Solution (Conditional Assembly, EXITM) 6-18
Third Solution (Nested Macro) 6-19

PART TWO: ASSEMBLER OPERATION

7. Assembler Overview

Versions of Assembler 7-1

MONITOR Assembler Environment 7-1
ISIS-II Assembler Environment 7-2

Input/Output Files 7-2

Source File 7-2
Object File 7-2
List File 7-2
Symbol-Cross-Reference File 7-3
ISIS-II Assembler Reserved File Names 7-3

8. Assembler Controls

Introduction to Assembler Controls 8-1

Primary and General Controls 8-1
Specifying Controls 8-1
Summary of Controls 8-2

ISIS-II Assembler Controls 8-2

ISIS-II Assembly-Time Command 8-2
Primary Controls 8-3
General Controls 8-5
Defaults 8-6
ISIS-II Embedded Control Lines 8-6

Intellec MONITOR Assembler Controls 8-7

MONITOR Assembly-Time Commands 8-7
Primary Controls 8-7
General Controls 8-8
Defaults 8-9
MONITOR Embedded Control Lines 8-9

9. Assembler Operation

ISIS-II Assembler Operation 9-1

Activation Sequence 9-1
Sample Assembly 9-1

Intellec MONITOR Assembler Operation 9-4

Activation Sequence 9-4
Sample Assembly 9-6

Appendix A: MCS-48 and UPI-41 Instruction Summary

Special Operators and Reserved Words A-1

MCS-48 and UPI-41 Assembly Language Notation A-3

Summary by Mnemonic Opcode A-4

Summary by Hexadecimal Opcode A-12

Appendix B: Assembler Directive Summary

Notation B-1

Summary of Directives B-2

Macro Directives B-3

Appendix C: Assembler Control Summary

Control Formats C-1

Common Controls C-1

ISIS-II Assembler Controls C-3

MONITOR Assembler Pass Options C-4

Defaults C-4

Appendix D: List File Formats

Assembly Listing Format D-1

Page Header D-1
Title Line D-2
Column Heading D-2
Assembly Output Line D-2
Symbol Table Listing D-3
Error Summary D-4

Symbol-Cross-Reference Listing D-5

Page Header D-5
Cross-Reference Output Line D-5

Appendix E: Reference Tables

ASCII Codes E-1

Powers of Two E-3

Powers of 16 (in Base 10) E-4

Powers of 10 (in Base 16) E-4

Hexadecimal-Decimal Integer Conversion E-5

Appendix F: Error Messages

Error Detection and Reporting F-1

Error Codes F-1

Source-File Errors F-1
Run-Time Errors F-3
Assembler Control Errors F-3
ISIS-II Error Messages F-4

ILLUSTRATIONS

Figure	*Title*	*Page*
1-1	Program Memory Map	1-3
1-2	Resident Data Memory Layout	1-4
1-3	Stack Format	1-6
1-4	Program Status Word Format	1-9
1-5	BUS Contents During Status Polling	1-12
2-1	Number Representation	2-4
3-1	Bit Rotation	3-21

PART ONE

PROGRAMMING THE MCS-48 AND UPI-41 MICROCOMPUTER FAMILIES

1. Functional Overview
2. Assembler Concepts
3. MCS-48 Assembly Language Instructions
4. UPI-41 Assembly Language Instructions
5. Assembler Directives
6. Macros

1. FUNCTIONAL OVERVIEW

A microcomputer, until recently, could be defined as a complete computer on a single board. At the heart of the microcomputer was the microprocessor device, or central processing unit (CPU). The board also contained control circuitry, memory devices, and input/output (I/O) interfaces.

The MCS-48 and UPI-41 microcomputer systems have made this traditional distinction between microcomputers and microprocessors obsolete. At the heart of these systems are several single-device microcomputers, each consisting of a CPU, separately addressable program and data memories, I/O interfaces, and timer. The systems are completed by the addition of applicable Intel peripherals, providing an extensive assortment of family parts. The MCS-48 microcomputer options are implemented as primary controllers of your OEM equipment. UPI-41 devices are implemented as intelligent, programmable peripheral processors.

The MCS-48 is available in six functionally similar versions – the 8048 and 8049 microcomputers with read-only (ROM) program memory, the 8748 microcomputer with erasable and programmable ROM (EPROM), the 8035 and 8039 microcomputers, which use no resident program memory, and the 8021 microcomputer, the lowest cost component in the MCS-48 family. The UPI-41 is based on either the 8041 microcomputer (with ROM program memory) or the 8741 microcomputer (with EPROM program memory). The following chart summarizes the main hardware differences among all eight microcomputers.

Microcomputer	*Pins*	*ROM*	*EPROM*	*RAM*	*External Addressing*
8048	40	1K	––	64	Yes
8748	40	––	1K	64	Yes
8035	40	––	––	64	Yes
8049	40	2K	––	128	Yes
8039	40	––	––	128	Yes
8021	28	1K	––	64	No
8041	40	1K	––	64	No
8741	40	––	1K	64	No

These hardware features are discussed in greater detail in the rest of this chapter.

The 8048, 8748, and 8035 are equivalent except for their program memories (ROM/EPROM). The 8035 is used with external program memories in prototype and preproduction systems. The 8049 and 8039 are also equivalent, except for program memory, and have the same instruction set as the 8048 group. For the purposes of this manual, which emphasizes programming primarily, '8048' refers to all five microcomputers.

Because of their different usage of the external bus, the 8041, 8741, and 8021 have a slightly different instruction set and functional approach from the other five. These differences are discussed at the end of this chapter and in Chapters 3 and 4. For the purposes of this manual, '8041' also refers to the 8741. Descriptions of the 8048 apply to the 8041, 8741, and 8021 also, except for specifically stated differences.

8048 BASIC FEATURES

From the programmer's viewpoint, the following are the main 8048 device features:

- Resident 2K or 1K by 8-bit ROM/EPROM program memory with memory expansion capability
- 128 or 64 by 8-bit random access (RAM) data memory, which includes the working registers and program counter stack and is also expandable
- 12-bit program counter (PC)
- Program status word (PSW), consisting of status bits, flags, and the stack pointer
- Programmable resident interval timer, also available as an external event counter
- Resident clock and oscillator for internal timing
- External and timer overflow interrupts
- I/O ports and controls, expandable using the 8243 expander device

Program Memory

Resident program memory consists ot a 2K by 8-bit ROM (8049) or a 1K by 8-bit ROM (8048) or EPROM (8748) divided into 256-byte 'pages.' In a typical development sequence, you might program the 8748 with your prototype code, debug this code using the Intellec system and ICE-48 facilities, and then commit the final version of your program to the 8048 ROM version for production. Or, you might prefer to use the 8748 for production, leaving yourself the option to make modifications in the field or to tailor your basic program to customer specifications.

Resident program memory can be expanded up to 4K using additional ROM or EPROM devices. This external memory is directly addressable by the 8048's 12-bit program counter. Address selection is done on a 'bank' basis using the MCS-48 instructions:

```
SEL MB0     ;SELECT MEMORY BANK 0
SEL MB1     ;SELECT MEMORY BANK 1
```

Memory bank 0 is the lower 2K of program memory and memory bank 1 is the upper 2K (Figure 1-1). Bits 0-10 of the program counter can address up to 2K locations; PC bit 11 is set to 1 by the SEL MB1 instruction, permitting addressing to 4K. The SEL MB instructions do not affect PC bit 11 until a branch from the main program sequence is executed (via a call or jump instruction).

NOTE

Program memory expansion beyond 4K is described in the MCS-48 user's manual. Program memory addressing using the EA (external address) pin is described in the same document.

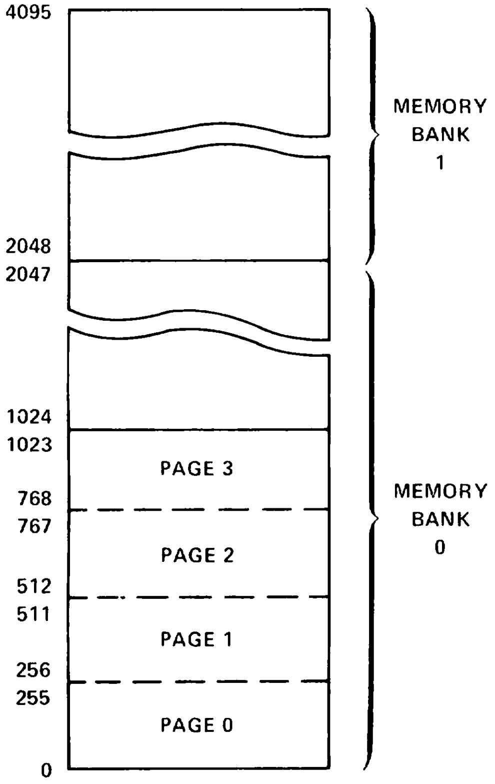


Figure 1-1 Program Memory Map

Data Memory

In addition to resident program memory, the 8049/8039 microcomputers provide a resident 128 by 8-bit data memory (expandable by 256 locations using additional RAM devices). The other MCS-48 microcomputers have a 64 by 8-bit resident data memory.

The memory consists of eight working registers (plus an additional eight registers selectable on a 'bank' basis), an eight-level program counter stack, and scratchpad memory (Figure 1-2). The amount of scratchpad memory available can vary depending on the number of addresses nested in the stack and the number of registers selected.

NOTE

Data memory expansion beyond 256 locations is described in the MCS-48 user's manual.

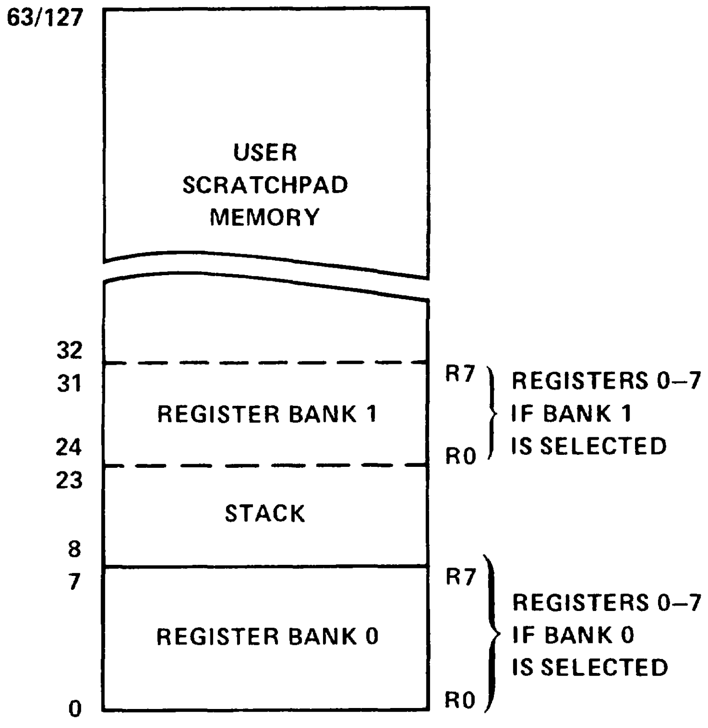


Figure 1-2 Resident Data Memory Layout

Addressing Data Memory

Working registers in RAM memory can be addressed 'directly' by specifying a register number, as in the instruction

```
MOV A,R4     ;MOVE THE CONTENTS OF REGISTER 4
             ;INTO THE ARITHMETIC AND LOGIC
             ;UNIT'S 8-BIT ACCUMULATOR
```

Other locations in resident data memory are addressed 'indirectly' using register 0 or register 1 to specify the addressed location. The special symbol '@' (commercial at) indicates that indirect addressing is desired.

```
MOV A,R1     ;MOVE THE CONTENTS OF REG 1 INTO THE
             ;ACCUMULATOR
MOV A,@R1    ;MOVE THE CONTENTS OF THE LOCATION WHOSE
             ;ADDRESS IS SPECIFIED BY REG 1 INTO THE
             ;ACCUMULATOR
```

Because all 128/64 locations (including the eight working registers) can be addressed by 7/6 bits, the most significant bits (6 and/or 7) of the addressing registers are ignored. However, all eight bits of register 0 or register 1 can be used in combination with the 8048's MOVX instructions to address up to 256 locations in external RAM data memory.

```
MOVX @R0,A   ;MOVE THE CONTENTS OF THE ACCUMULATOR
             ;INTO THAT LOCATION IN EXTERNAL DATA
             ;MEMORY WHOSE ADDRESS IS CONTAINED
             ;IN REGISTER 0
```

Working Registers

The dual bank of eight working registers is selected by the 8048's SEL RB instruction. The initial setting is 'bank 0,' which refers to data memory locations 0-7. If the instruction

```
SEL RB1      ;SELECT REGISTER BANK 1
```

has been issued, then references to R0-R7 in MCS-48 instructions operate on locations 24-31. As was mentioned above, registers 0 and 1 in the active bank have a special addressing function; they are used to address indirectly all locations in scratchpad memory (including the optional 256-location expansion). These indirect RAM address registers are especially useful for repetitive operations on adjacent data memory locations, as in the following example:

```
START:     ADD A,@R0  ;ADD TO THE ACCUMULATOR THE
                      ;CONTENTS OF THE LOCATION
                      ;WHOSE ADDRESS IS SPECIFIED
                      ;BY REG 0
           INC R0     ;INCREMENT REG 0
           JNC START  ;JUMP TO INSTRUCTION LABELED
                      ;'START' IF NO ADDITION
                      ;OVERFLOW (NO CARRY)
```

A good programming practice is to reserve locations 24-31 for interrupt servicing routines, thereby preserving the contents of your main program registers. Simply specify SEL RB1 as one of your interrupt routine's initialization instructions. When you subsequently return to the main program using the instruction RETR, the previously selected bank is automatically restored. During interrupt processing, registers in bank 0 can be accessed indirectly.

Unused registers can serve as additional scratchpad memory, if desired.

Program Counter Stack

Locations 8-23 are used as an 8-level program counter stack. When control is temporarily passed from the main program to a subroutine or interrupt servicing routine, the 12-bit program counter and bits 4-7 of the program status word (PSW) are stored in two stack locations (Figure 1-3). Note that the program counter is stored with its low-order bits in the lowest available address in the stack area.

When control returns to the main program via an RETR instruction, the program counter and PSW bits 4-7 are restored. Returning via an RET instruction does not restore the PSW bits, however. (These PSW bits are described in detail later in this chapter.)

The program counter stack is addressed by three stack pointer (STP) bits in the PSW (bits 0-2). The *current* program counter is not resident in the program counter stack and consequently is not directly accessible.

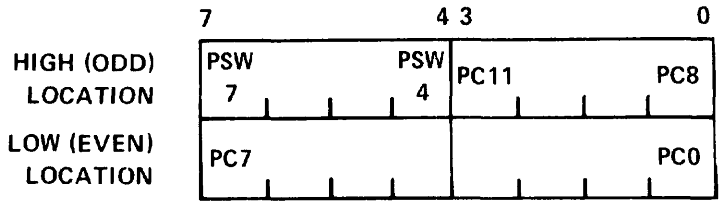


Figure 1-3 Stack Format

The stack pointer bits in the PSW refer to the stack pointer locations as follows:

STP Bits	*Data Memory Locations*
000	8-9
001	10-11
010	12-13
011	14-15
100	16-17
101	18-19
110	20-21
111	22-23

The bit setting indicates the locations to be loaded the next time the program counter is stored. The stack pointer is incremented by one each time the program counter is stored and decremented each time the program counter is restored. Unused stack locations can be employed as scratchpad memory.

The 8048 stack allows up to eight levels of subroutine 'nesting;' that is, a subroutine may call a second subroutine, which may call a third, etc., up to eight levels. When processing interrupts, remember that the stack contains not only information nested by the main program, but also the program counter stored by the interrupt, plus any information required by subroutine nesting in the interrupt service routine.

Programmable Controls

The 8048 provides several condition bits, flags, and pins for testing and controlling program operation. These are referred to as:

C	Carry bit
AC	Auxiliary carry bit
F0	Flag 0
F1	Flag 1
BS	Register bank switch
T0	Test 0 pin
T1	Test 1 pin
TF	Timer flag
I	Interrupt input pin

Carry Bit

The carry bit (C) is affected by the addition and decimal adjust instructions and certain rotate operations and generally indicates a carry out of the bit 7 position (most significant bit, or MSB) of the 'accumulator' (ACC – a special register in the 8048's arithmetic and logic unit). For example, addition of two 8-bit numbers as in the following instructions would result in a carry out of the MSB and set the carry bit.

```
MOV A,#0AEH   ;MOVE VALUE 'AE' HEX TO ACC
ADD A,#74H    ;ADD VALUE '74' HEX TO ACC
```

Bit	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
AE	1	0	1	0	1	1	1	0
+74	0	1	1	1	0	1	0	0
=122	0	0	1	0	0	0	1	0
	1		Carry					

The carry bit can be complemented (changed to 0 if 1, or to 1 if 0) using the MCS-48 instruction CPL C, reset to zero using CLR C, and tested by the conditional jump instructions JC and JNC.

Auxiliary Carry Bit

The auxiliary carry (AC) bit indicates a carry out of bit 3 in the accumulator and is only applicable when decimal arithmetic is being performed. This bit essentially allows the Decimal Adjust Accumulator (DA A) instruction to perform its function. The DA instruction adjusts the 8-bit accumulator value to form two 4-bit Binary-Coded-Decimal (BCD) digits. The following instruction sequence resets the carry bit to zero and sets the auxiliary carry bit.

```
MOV A,#2EH    ;MOVE VALUE '2E' HEX TO ACC
ADD A,#74H    ;ADD VALUE '74' HEX TO ACC
```

Bit	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
2E	0	0	1	0	1	1	1	0
+74	0	1	1	1	0	1	0	0
=A2	1	0	1	0	0	0	1	0
	0 Carry				1 Auxiliary Carry			

The auxiliary carry bit cannot be tested or altered directly (but see the discussion of the PSW later in this chapter). It is affected only by addition.

Flag Bits (F0, F1)

The 8048 provides two program control flags (F0 and F1), both of which can be complemented with the instructions CPL F0/F1, reset to zero using CLR F0/F1, or tested with the conditional jumps JF0 and JF1. Their initial state is zero.

One important difference between these two flags is that F0 is restored when control is returned from an interrupt servicing routine (by the RETR instruction), whereas F1 is not. Therefore the latter can be used by the interrupt servicing routine to pass an information bit to the main program.

Register Bank Switch

The register bank switch indicates which of the possible register banks (0 or 1) is active. It is toggled by the 8048 instructions SEL RB0 and SEL RB1. Its initial state is zero.

Test Input 0

Test input 0 (T0) provides a multifunction capability for the design engineer and programmer. It is directly testable using the MCS-48 conditional jump instructions JT0 and JNT0.

As an input pin activated by an external source it could be used as a pseudo interrupt or other general-purpose function.

T0 can also be converted to a state clock output using the MCS-48 instruction ENT0 CLK. This signal could then be used as a general-purpose clock by the MCS-48. (See the MCS-48 user's manual for details.)

Test Input 1

A special 8048 register can be used as an interval timer or as an external event counter. As an interval timer it is initiated by the STRT T instruction and incremented by a prescaler having a periodic duration equivalent to 32 instruction cycles (at 2.5 microseconds per cycle). When the register is used as an event counter, the prescaler is bypassed and the external test 1 (T1) pin is designated as the counter input. The latter mode is enabled by the STRT CNT instruction. Both modes are disabled by the STOP TCNT instruction. The conditional jump instructions JT1 and JNT1 can be used to test this pin.

Timer Flag

As was mentioned above, the interval timer is incremented every 32 instruction cycles. This means the 8-bit timer register will overflow every 8192 cycles (256 x 32). When the timer overflows, the timer flag (TF) is set, whether or not the timer overflow interrupt is enabled. The same is true of an event counter overflow (more than 255 T1 inputs).

The timer flag can be tested by the conditional jump instruction JTF. It is reset to zero each time this instruction is executed. Its initial state is zero.

Interrupt Input Pin

If the 'external' interrupt is enabled and this pin is active low (zero level), an interrupt is initiated. (See the discussion of interrupts below.)

The MCS-48 conditional jump instruction JNI tests for the zero level at this pin. With the interrupt disabled, this instruction could be used as another test input.

Program Status Word

The program status word (PSW) consists of eight bits organized as shown in Figure 1-4.

7							0
C	AC	F0	BS	1	SP2	SP1	SP0

Figure 1-4 Program Status Word Format

As this figure indicates, locations 4-7 contain the register bank switch (BS), flag 0 (F0), auxiliary carry bit (AC), and carry bit (C). These four bits are stored in the stack with the program counter when a CALL instruction or an interrupt is encountered. The bits are restored by an RETR return instruction (but not by RET).

Bits 0-2 of the PSW contain the stack pointer (STP) used to address the 8-level data memory stack (see the subsection 'Program Counter Stack', above). Bit 3 of the PSW is unused and is always set to one.

Two MCS-48 instructions (MOV A,PSW and MOV PSW,A) allow data to be transferred between the PSW and the accumulator. This is particularly useful for modifying the stack pointer or AC bits. Bits 4, 5, and 7 can also be modified individually using the instructions mentioned above (for example, SEL RB1, CLR F0, CPL C).

Interrupts

The 8048 responds to two kinds of interrupts: 'external' and 'timer overflow.' An external interrupt forces a call to location 3 in program memory; a timer overflow interrupt forces a call to location 7.

The external interrupt is enabled by the instruction EN I and disabled by the instruction DIS I. If this interrupt is enabled and the interrupt input pin goes low (level zero), the interrupt sequence is initiated as soon as the currently executing instruction is completed. A CALL to location 3 is forced, the return address and bits 4-7 of the PSW are stored in the program stack, and the stack pointer bits incremented. If you wish, you can create your own 'interrupt acknowledge' by programming an appropriate output pin or by implying the acknowledge in ensuing I/O operations.

The RETR instruction should be used to return from an interrupt. This instruction will restore the program counter and PSW bits 4-7, providing automatic restoration of the previously-active register bank as well. RETR also reenables interrupts.

The timer-overflow interrupt is enabled by the EN TCNTI instruction and disabled by the DIS TCNTI instruction. If enabled, this interrupt occurs when the timer/event-counter register overflows. A CALL to location 7 is forced and the interrupt routine proceeds as described above.

After an overflow the timer continues to accumulate time. If you require time intervals greater than the maximum, you can disable the interrupt, count the number of overflows using the JTF (JUMP if timer flag is one) instruction, and accumulate the number of overflows in a software counter until the required time is reached. Note that reading the timer flag with a JTF resets it to zero.

While an interrupt service routine is executing, new timer interrupt requests will be accepted, but they cannot be serviced until the current routine is completed. New external interrupts are not saved. If an external interrupt and a timer-overflow interrupt occur simultaneously, both are recognized but the external interrupt has highest priority.

NOTE

> All routines for servicing interrupts must be located in memory bank 0 (program memory locations 0-2K). During servicing of an interrupt, PC bit 11 is held at zero. The SEL MB (select memory bank) instructions should not appear in an interrupt service routine.

Input/Output

Of the 40 pins on the 8048, 27 can be used for input, output, or both, depending on the MCS-48 configuration established. In addition to the I/O capability provided by these pins, the 8243 expander device can be added to the configuration to provide 16 additional I/O lines (four 4-pin ports).

NOTE

> I/O expansion beyond that provided by a single 8243 expander device is described in the MCS-48 user's manual.

The total 43 I/O lines possible with an 8048 and 8243 expander device are divided into eight directly addressable groups as follows:

Port	*Pins*	*Comment*
BUS	D0-D7	Bidirectional. Strobed input.
1	P10-P17	Quasi-bidirectional depending on configuration.
2	P20-P27	P20-23 are used to attach four 8243 ports. Quasi-bidirectional.
--	T0, T1, $\overline{\text{INT}}$	Testable input pins; test 0, test 1, interrupt.
4-7	0-15	Four pins each. 8243 ports

The BUS port and ports 1-2 on the 8048 and ports 4-7 on the 8243 can be read and written by 8048 I/O instructions. The BUS and ports 1-2 can be ANDed and ORed with the second byte of ANL and ORL instructions.

For example:

```
ANL BUS,#data        ;'AND' SECOND BYTE WITH DATA IN
                     ;BUS PORT
ORL P2,#data         ;'OR' SECOND BYTE WITH DATA IN
                     ;PORT 2
```

Ports 4-7 can be ANDed and ORed with the low-order four bits of the accumulator.

```
        ORLD P5,A           ;'OR' ACC BITS 0-3 WITH DATA
                            ;IN PORT 5
```

Address and control data are provided to the 8243 ports via 8048 pins P20-23. Any data existing on P20-23 before an 8243 instruction is issued is lost. Therefore, if your configuration includes an 8243 expander device, pins P20-23 should not be used for general I/O operations.

UPI-41 MICROCOMPUTERS

The 8041 and 8741 (UPI-41) microcomputers are variations of the 8048 and 8748, respectively. The essential difference between the 8041 and 8048 is that the 8041 includes handshaking interfaces and protocols for MCS-48, MCS-80, and MCS-85 buses, enabling it to serve as a programmable, intelligent peripheral within a larger microcomputing system. This section focuses on the specific design and functional differences between the 8041 and the 8048 required to implement this handshaking. Differences in the assembly language instructions for these devices are described in Chapter 4.

Functional Differences

During the transfer of data between a master computer and the 8041, the handshaking protocol requires the 8041's BUS port for interfacing to the master port. As a consequence, 8041 program memory cannot be expanded beyond 1K and data memory cannot be expanded beyond 64 locations. I/O can still be expanded using the 8243 expander device, however.

The external interrupt function is also committed to the master processor interface. However, the event counter can provide an effective external interrupt if it is preset to all ones. The T1 input can then be used in the same manner as the interrupt input, but program control is passed to location 7 rather than location 3 in this instance.

```
    MOV A,#0FFH  ;MOVE 'ONES' TO ACC
    MOV T,A      ;MOVE ACC DATA TO TIMER
    EN TCNTI     ;ENABLE COUNTER INTERRUPT
    STRT CNT     ;START EVENT COUNTER
```

In 8041 mode the EN I and DIS I instructions used to enable/disable external interrupts on the 8048 have a different function. When the master processor is transferring data to the 8041 slave, it can cause an interrupt each time it fills the 8041's data bus buffer (described below) to ensure that two writes are not issued before the buffer is cleared. EN/DIS I enable and disable this interrupt. When initiated, this interrupt passes control to location 3 as in the normal 8048 external interrupt procedure.

When data is transferred from the 8041 to the master computer, no interrupt is possible except by dedicating I/O lines. The master must poll special 8041 status bits (described below) to determine whether the data bus buffer is empty.

Finally, the T0 pin can be used only as a test input in 8041 mode; it cannot be used as a state clock output.

Hardware Differences

Hardware differences (such as pin designation differences, deletion of the functions described above, and handshaking hardware) are described in detail in the UPI-41 user's manual. However, two special 8041 registers used in these protocols should be singled out since they are referenced in 8041 instructions.

Data Bus Buffer

The 8-bit data bus buffer (DBB) serves as a temporary register for information flowing between the 8041 and a master computer. Transfers between the master and slave processors via the data bus buffer can be implemented with or without program interference (using EN I or DIS I).

Data is transferred between the DBB and the 8041's accumulator using the UPI-41 instructions:

```
IN A,DBB        ;PLACE DBB CONTENTS INTO 8041 ACC
OUT DBB,A       ;PLACE 8041 ACC CONTENTS INTO DBB
```

Status Register

This 4-bit register indicates the status of flag 0 and flag 1 (F0 and F1) and of two special 8041 flags; input buffer (IBF) and output buffer (OBF). IBF and OBF indicate the condition of the data bus buffer and are initially cleared.

The sequence for transferring data from a master processor to the 8041 is as follows:

- Eight bits are written from the BUS port into the 8041's DBB
- IBF is set
- Control/data input is placed in flag 1 (F1)
- An interrupt is generated, if enabled

Subsequent execution of the UPI-41 instruction IN A,DBB in either the main program or the interrupt service routine clears IBF. The master can determine that IBF has been cleared (that is, DBB is empty and ready for more data) by polling the status register. A 'read control status' pulse places the 4-bit status register and 4 undefined high-order bits on the BUS in the order shown in Figure 1-5.

D7							D0
–	–	–	–	F1	F0	IBF	OBF

Figure 1-5 BUS Contents During Status Polling

When an OUT DBB,A instruction is executed in a UPI-41 program, initiating a transfer of data from the slave to the master computer, OBF is set. A subsequent 'read data bus buffer' pulse from the master reads the DBB contents onto the BUS and clears OBF.

The slave computer cannot poll or interrupt the master, but it can check the status of the DBB using the two UPI-41 instructions:

```
JNIBF  addr ;JUMP TO 'ADDR' IF IBF NOT SET
JOBF   addr ;JUMP TO 'ADDR' IF OBF SET
```

8021 MICROCOMPUTER

The 8021 is the low-cost, low-end product within the MCS-48 family. It's features are a subset of the 8048 features described earlier in this chapter. Consequently, a number of instructions in the 8048 instruction set are not applicable to the 8021 (see Chapter 3 and Appendix A).

Functional Differences

The fundamental difference between the 8048 and 8021 is in packaging (28 pins on the 8021 vs. 40 pins on the 8048) and the absence of the BUS port.

The fewer number of pins results in fewer programmable controls and interrupts. The 8021 does contain its own inboard oscillator, however, and provides the same timer/event-counter capability as the 8048 (using the T1 test input pin and TF timer flag).

The absence of the BUS port means the 1K on-chip ROM memory and 64-byte RAM memory cannot be expanded, and 8048 instructions referencing expanded memory are not applicable. Of the 28 pins on the 8021 package, 20 are available for I/O, including I/O expansion using the 8243 expander device. The 20 I/O lines possible with an 8021 and 16 expander device lines are divided into the following directly addressable groups:

Port	*Pins*	*Comment*
0	P00-P07	Quasi-bidirectional with open drain outputs; optional pullup device deletion.
1	P10-P17	Quasi-bidirectional.
2	P20-P23	Quasi-bidirectional. Used to attach four 8243 ports.
–	T1	Testable input pin.
4-7	0-15	Four pins each. 8243 ports.

The 8021, like the 8048, provides eight directly-addressable registers (locations 0-7 in RAM memory). All locations (0-64) in RAM memory can be addressed indirectly through registers 0 and 1. Register bank selection is not available on the 8021.

Hardware Differences

Hardware differences between the 8048 and 8021 are described in the MCS-48 user's manual.

2. ASSEMBLER CONCEPTS

ASSEMBLERS AND ASSEMBLY LANGUAGE

If you have ever written a computer program in a machine-recognizable form such as binary code, you will be particularly appreciative of the advantages of programming in a symbolic assembly language. Assembly-language operation codes (opcodes) are easily remembered (for example, MOV for a 'move' instruction, JMP for a 'jump'). You can also express symbolically the addresses and values referenced in the operand field of assembly language instructions. The names for these operands can be selected to suggest their purpose, making them as mnemonic as the opcodes.

The program consisting of assembly language instructions is called a *source program.* This program is passed through an assembler, which performs the clerical task of translating symbolic code into *object code* recognizable by the MCS-48 and UPI-41 microcomputers.

The *source file* passed to the assembler actually includes more than *source program* instructions. It also includes *assembler directives* and (possibly) *assembler controls.* Only source program instructions are converted into executable object code, however. The assembler directives and controls initiate various functions that assist and direct the assembler in its translation operation.

| The diskette-resident 8048/8041 assembler, in addition to allowing symbolic programming, is also a *macro assembler.* Frequently repeated routines, identical except for certain parameters, need be coded only once and thereafter can be generated by a single instruction containing the specific parameters needed. Such routines are called *macros.* Macro definition is described in detail in Chapter 6.

Assembler output consists of three possible files: the *object file* containing your program code in machine-executable form, the *list file* printout of your source code, object code, and symbol table, and the *symbol-cross-reference file,* a listing of symbol-cross-reference records. These files are discussed more fully in Part Two.

In this chapter, references to the MCS-48 instruction set apply to the UPI-41 instruction set as well.

INSTRUCTION FORMAT

MCS-48 assembly-language instructions and assembler directives consist of up to four fields as follows:

```
Label:    Opcode    Operand, Operand    ;Comment
```

The label and comment fields are always optional. The operand field may contain zero, one, or two operands depending on the opcode specified. Any number of blanks can separate fields. The entire instruction must be entered on one line, terminated by a carriage return and line feed. No continuation lines are possible, though you may have lines consisting entirely of comments.

Label Field

An instruction label is a symbol name whose value is the specific memory location where the instruction resides. It is optional and when present must be followed by a colon. A label can be one to six alphanumeric characters, with the first character alphabetic. A symbol used as a label cannot be redefined elsewhere in your program. (See 'Symbols and Symbol Tables' later in this chapter.)

Opcode Field

This field contains the mnemonic operation code for the MCS-48 instruction or assembler directive to be performed. It is terminated by a blank or nonalphanumeric character, or by a carriage return and line feed if no operand or comment field is present.

Operand Field

The operand field identifies the data to be operated on by the specified instruction opcode. Some instructions require no operand. Others require one or two operands. In the latter case, the operands are separated by a comma. As a general rule, *when two operands are required (data transfer, addition, and logical operations), the first operand specifies the destination (or target) of the operation's result and the second operand specifies the source data.*

```
ADD A,R3          ;ADD CONTENTS OF REG 3 TO ACC
ANL A,R3          ;LOGICAL 'AND' CONTENTS OF ACC
                  ;WITH MASK CONTAINED IN REG 3
MOV R1,#0FFH      ;MOVE 'FF' HEX (ONES) INTO REG 1
```

Operands can reference directly data contained in MCS-48 registers such as the PSW, accumulator, or data memory working registers 0-7.

```
MOV A,PSW         ;MOVE PSW CONTENTS TO ACC
XCH A,R4          ;EXCHANGE ACC DATA WITH
                  ;REG 4 DATA
```

All data memory locations can be accessed indirectly by prefacing a reference to Register 0 or 1 with a 'commercial at' sign (@).

```
MOV @R0,A         ;MOVE ACC CONTENTS TO DATA MEMORY
                  ;LOCATION WHOSE ADDRESS IS
                  ;SPECIFIED IN REG 0
```

The JMPP instruction allows program memory locations to be accessed indirectly by prefacing an accumulator reference with @.

```
JMPP @A           ;CONTENTS OF PROGRAM MEMORY LOCATION POINTED TO BY
                  ;ACC ARE SUBSTITUTED FOR BITS 0-7 OF PROGRAM COUNTER
```

Operands can contain 'immediate' data. The desired value is inserted directly into the operand field. All immediate data must be prefixed with a pound sign (#) to distinguish it from register data and must evaluate to eight bits.

Immediate data can be in the form of an ASCII constant (a character enclosed in single quotes), a number, an expression to be evaluated at assembly time, or a symbol name. To indicate a quote as an ASCII constant, show the quote as two consecutive single quotes (''). Any symbol appearing in the operand field must be previously defined.

```
MOV A,#'T'          ;MOVE THE VALUE OF ASCII
                    ;CONSTANT 'T' (01010100)
                    ;INTO ACC
ADD A,#0AH          ;ADD HEX '0A' (00001010)
                    ;TO ACC
ANL A,#3+(D/5)      ;LOGICAL 'AND' CONTENTS OF
                    ;ACC WITH MASK WHOSE VALUE
                    ;IS THE RESULT OF '3+(D/5)'
```

Finally, the operand field of a jump instruction (that is, the address to be jumped to) can be expressed as a symbolic label, as an absolute 12-bit program memory address, or as an expression that can be evaluated to such an address. In no case is this operand preceded by a pound sign.

```
JMP START      ;JUMP TO THE LOCATION LABELED 'START'
JMP 200H       ;JUMP TO LOCATION 200 HEX (512 DECIMAL)
```

Expression evaluation and symbols are discussed in more detail in the next two sections of this chapter.

Comment Field

The comment field can contain any information you deem useful for annotating your program. The only stipulation is that this field be preceded by a semicolon. A double semicolon (;;) preceding a comment in the body of a macro definition suppresses inclusion of the comment in the macro definition, thus reducing storage requirements.

ARITHMETIC OPERATIONS

When discussing arithmetic operations, we must distinguish between operations performed by your program when it is executed (such as ADD A,R5) and expression evaluation performed by the assembler at assembly time (such as MOV A,#P+3*(X/2). Numbers are represented identically in both cases, but your program has considerably more flexibility than the assembler in determining the range of numbers, internal notation, and whether numbers are to be considered signed or unsigned. The characteristics of both modes of arithmetic are summarized in Figure 2-1 and discussed in more detail in the following subsections.

Number Characteristic	Assembly—Time Expression Evaluation	Program Execution Arithmetic
Base Representation	Binary, Octal, Decimal, or Hexadecimal	Binary, Octal, Decimal, or Hexadecimal
Range	0-(64K-1)	User Controlled
Evaluates To:	16 Bits	User Interpretation
Internal Notation	Two's Complement	Two's Complement
Signed/Unsigned Arithmetic	Unsigned	Unsigned Unless User Manipulates

Figure 2-1 Number Representation

Number Base Representation

Numbers can be expressed in decimal, hexadecimal, octal, or binary form. A hexadecimal number must begin with a decimal digit and have the suffix 'H' (for example: 3AH, 0FFH, 12H). Octal values must have one of the suffixes 'O' or 'Q' (for example: 76O,53Q). Binary numbers must have the suffix 'B' (for example: 10111010B). Decimal numbers can be suffixed optionally by 'D' (for example: 512, 512D). Where no suffix is present, decimal is assumed.

Permissible Range of Numbers

In general, numbers can range between 0 and 65,535 (0FFFFH). Numbers outside this range are evaluated 'modulo' 64K (that is, a number greater than 64K is divided by 64K and the remainder substituted for the original number). All expressions can be evaluated to 16 bits.

Certain limitations must be applied within this general range, however. For example, most program execution arithmetic is done using the 8-bit accumulator or 8-bit registers and most results evaluate to 8 bits. To work with larger numbers would require manipulation of register pairs.

If you are doing signed arithmetic, the high-order bit of each number is used to indicate the sign of that number (0 if positive, 1 if negative). Consequently, the remaining bits can only express a number in the range —32,768 to +32,767 for 16-bit arithmetic. For 8-bit arithmetic, the range is —128 to +127.

If a number is too large for its intended use, either an error results or modulo arithmetic is performed. For example:

- Program memory addresses must be in the range 0-4095 (12 bits). In some cases, an address reference must be 'within page,' that is, within the range 0-255 (8 bits).
- Data memory addresses must be in the range 0-255 (8 bits).

- Operands containing 8-bit immediate data must evaluate to an 8-bit number.
- Expressions in a DB assembler directive (except strings) must evaluate to 8 bits.

Two's Complement Arithmetic

Two's complement notation allows subtraction to be performed by a series of bit complementations and additions (thus reducing the circuitry requirements of a processor). A number is converted to two's complement form by complementing all its bits and adding a binary one to the result.

When a number is interpreted as a signed two's complement number, the low-order bits supply the magnitude of the number and the high-order bit is interpreted as the sign of the number. As was mentioned above, the range of a signed two's complement value is –32,768 to +32,767 (for 16 bits) and –128 to +127 (for 8 bits).

When a 16-bit value is interpreted as an unsigned two's complement number, it is considered to be positive and in the range 0-65,535. An 8-bit value is in the range 0-255.

The assemblers perform all expression evaluation assuming unsigned two's complement numbers. Similarly, execution-time arithmetic normally assumes unsigned two's complement notation, but you can perform signed arithmetic by isolating and inspecting the high-order bit with the instruction:

```
JB7 MINUS       ;IF ACC BIT 7=1 GO TO 'MINUS' ROUTINE
```

The MCS-48 instruction set does not include a subtraction instruction. Subtraction is done by complementing the accumulator and proceeding as in a normal two's complement addition operation. The CPL A (complement accumulator) instruction performs a straight binary one's complement. You must perform the binary addition of one, necessary to convert the number to two's complement notation, yourself.

Example: Subtract 1AH from 63H using signed two's complement notation.

```
MOV A,#1AH     ;MOVE '1AH' INTO ACC (00011010)
CPL A          ;ONE'S COMPLEMENT ACC (11100101)
INC A          ;CONVERT TO TWO'S COMPLEMENT
               ;(11100110)
ADD A,#63H     ;ADD '63' TO VALUE IN ACC (01001001)
JB7 MINUS      ;IF ACC BIT 7=1 GO TO 'MINUS' ROUTINE
```

The result is +49H.

Assembly–Time Expression Evaluation

An expression is a combination of numbers, symbols, and operators. The latter can be arithmetic, relational, and logical operators or specially-defined MCS-48 operators. Any symbol appearing in an expression must have a previously-defined absolute value.

The ASCII characters 'null' and 'rubout' are ignored on input, but the null string can be represented by two consecutive quotes or by a missing operand. The null string is illegal in any context that requires numerical evaluation.

Operators

The assembler includes five groups of operators that permit the following assembly-time operations: arithmetic, bit shifting operations, logical evaluation, value comparison, and byte isolation. These are all assembly-time operations. Once the assembler has evaluated an expression, it becomes a permanent part of your program.

Arithmetic Operators

The arithmetic operators are as follows:

Operator	*Meaning*
+	Unary or binary addition
—	Unary or binary subtraction
*	Multiplication
/	Division. Any remainder is discarded (7/3=2)
MOD	Modulo. Result is remainder produced by a division operation (7 MOD 3 = 1)

Examples:

The following expressions generate the bit pattern for the ASCII character A:

```
5+30*2
(25/5)+30*2
5 + (—30 * —2)
```

The MOD operator must be separated from its operands by spaces:

```
NUMBR MOD 8
```

Assuming that NUMBR has the value 25, this expression evaluates to 1.

Shift Operators

The shift operators are as follows:

Operators	*Meaning*
y SHR x	Shift operand 'y' to the right 'x' bit positions
y SHL x	Shift operand 'y' to the left 'x' bit positions

The shift operators do not wrap around any bits shifted out of the byte. Bit positions vacated by the shift operation are replaced with zeros. The shift operator must be separated from its operands by spaces.

Example:

Assume that NUMBR has the value 0101 0101. The effects of the shift operators is as follows:

NUMBR SHR 2	0001 0101
NUMBR SHL 1	1010 1010

Shifting one bit position to the left has the effect of doubling a value; a shift one bit position to the right has the effect of dividing a value in half.

Logical Operators

The logical operators are as follows:

Operator	*Meaning*
NOT	Logical one's complement
AND	Logical AND (=1 if both ANDed bits are 1)
OR	Logical OR (=1 if either ORed bit is 1)
XOR	Logical EXCLUSIVE OR (=1 if bits are different)

The logical operators act only upon the least significant bit of values involved in the operation. Also, these operators are commonly used in conditional IF directives. These directives are fully explained in Chapter 5.

Example:

The following IF directive tests the least significant bit of three items. The assembly language code that follows the IF is assembled only if the condition is TRUE. This means that all three fields must have a one bit in the least significant bit position.

```
IF FLD1 AND FLD2 AND FLD3
          •
          •
          •
```

Compare Operators

The compare operators are as follows:

Operator	*Meaning*
EQ	Equal
NE	Not equal
LT	Less than
LE	Less than or equal
GT	Greater than
GE	Greater than or equal
NUL	Special operator used to test for null (missing) macro parameters. (ISIS-II assembler only.)

The compare operators yield a yes-no result. Thus, if the evaluation of the relation is TRUE, the value of the result is all ones. If FALSE, the value of the result is all zeros. Relational operations are based strictly on magnitude comparisons of bit values. Thus, a two's complement negative number (which always has a one in its high order bit) is greater than a two's complement positive number (which always has a zero in its high order bit).

Since the NUL operator applies only to the macro feature, NUL is described in Chapter 6.

The compare operators are commonly used in conditional IF directives. These directives are fully explained in Chapter 5.

Notice that the compare operator must be separated from its operands by spaces.

Example:

The following IF directive tests the values of FLD1 and FLD2 for equality. If the result of the comparison is TRUE, the assembly language coding following the IF directive is assembled. Otherwise, the code is skipped.

```
IF FLD1 EQ FLD2
      .
      .
      .
```

Byte Isolation Operators

The byte isolation operators are as follows:

Operator	*Meaning*
HIGH	Isolate high-order 8 bits of 16-bit value
LOW	Isolate low-order 8 bits of 16-bit value

As was mentioned in the discussion of number ranges, you will sometimes need an 8-bit address or need to generate an 8-bit value. This is where the HIGH and LOW operators can be useful.

Example:

Assume ADRS is an address manipulated at assembly-time for building tables or lists of items that must all be below address 255 in memory. The following IF directive determines whether the high-order byte of ADRS is zero, indicating the address is still less than 256:

```
IF HIGH ADRS EQ 0
        .
        .
        .
```

Precedence of Operators

Expressions are evaluated left to right. Operators with higher precedence are evaluated before other operators that immediately precede or follow them. When two operators have equal precedence, the leftmost is evaluated first.

Parentheses can be used to override normal rules of precedence. The part of an expression enclosed in parentheses is evaluated first. If parentheses are nested, the innermost are evaluated first.

$$15/3 + 18/9 \quad = \quad 5 + 2 = 7$$

$$15/(3 + 18/9) = \quad 15/(3 + 2) = 15/5 = 3$$

The following list describes the classes of operators in order of precedence:

- Parenthesized expressions
- NUL
- HIGH, LOW
- Multiplication/Division: *, /, MOD, SHL, SHR
- Addition/Subtraction: +, – (Unary and binary)
- Relational Operators: EQ, LT, LE, GT, GE, NE
- Logical NOT
- Logical AND
- Logical OR, XOR

The relational, logical, and HIGH/LOW operators must be separated from their operands by at least one blank.

SYMBOLS AND SYMBOL TABLES

Symbolic Addressing

If you have never done symbolic programming before, the following analogy may help clarify the distinction between a 'symbolic' and an 'absolute' address.

The locations in program memory can be compared to a cluster of post office boxes. Suppose Richard Roe rents box 500 for two months. He can then ask for his letters by saying 'Give me the mail in box 500,' or 'Give me the mail for Roe.' If Donald Doe later rents box 500, he too can ask for his mail by either box number 500 or by his name.

The content of the post office box can be accessed by a fixed, *absolute* address (500) or by a *symbolic, variable* name. The postal clerk correlates the symbolic names and their absolute values in his log book. The MCS-48 clerk, the assembler, performs the same function, keeping track of symbols and their values in a *symbol table.* Note that you do not have to assign values to symbolic addresses. The assembler references its location counter during the assembly process to calculate these addresses for you. (The location counter does for the assembler what the program counter does for the microcomputer. It tells the assembler where the next instruction or operand is to be placed in memory.)

Symbol Characteristics

A symbol can contain one to six alphabetic (A-Z) or numeric (0-9) characters (with the first character alphabetic) or the special character '?'. A dollar sign can be used as a symbol to denote the value currently in the location counter. For example, the command

```
JMP $+6
```

forces a jump to the instruction residing six memory locations higher than the JMP instruction. Symbols of the form '??nnnn' are generated by the assembler to uniquely name symbols local to macros.

The assemblers regard symbols as being reserved or user-defined, global or limited, permanent or redefinable. All MCS-48 symbols are absolute, that is, fixed to some absolute memory address or fixed-value expression unaffected by program loading.

Reserved, User-Defined, and Assembler-Generated Symbols

The '$' symbol and following MCS-48 and UPI-41 instruction-set opcodes are reserved and cannot be specified as user-defined symbols except in a limited context (as macro dummy parameters or as symbols defined as local to a macro definition).

ADD
ADDC
ANL
ANLD
CALL
CLR
CPL
DA
DEC
DIS
DJNZ
EN
ENT0
IN
INC
INS
JBn
JC
JF0
JF1
JMP
JMPP
JNC
JNI
JNIBF
JNT0
JNT1
JNZ
JOBF
JTF
JT0
JT1
JZ
MOV
MOVD
MOVP
MOVP3
MOVX
NOP
ORL
ORLD
OUT
OUTL
RET
RETR
RL
RLC
RR
RRC
SEL
STOP
STRT
SWAP
XCH
XCHD
XRL

The following instruction operand symbols and symbols required by the assembler are also reserved:

Symbol	*Meaning*
A	Accumulator
R0	Register 0
R1	Register 1
R2	Register 2
R3	Register 3
R4	Register 4
R5	Register 5
R6	Register 6
R7	Register 7
PSW	Program Status Word
BUS	BUS Port
P0	I/O Port 0 (8021)
P1	I/O Port 1
P2	I/O Port 2
P4	I/O Port 4
P5	I/O Port 5
P6	I/O Port 6
P7	I/O Port 7
C	Carry Flag
T	Timer Register
CNT	Counter Register
TCNT	Timer/Counter
RB0	Register Bank 0
RB1	Register Bank 1
MB0	Memory Bank 0
MB1	Memory Bank 1
I	Interrupt
TCNTI	Timer/Counter Interrupt
F0	Flag 0
F1	Flag 1
DBB	Data Bus Buffer (8041)
AN0	Assembler reserved operands
AN1	
FLAGS	
RAD	
STS	

Finally, the following directives cannot be used as symbols except in a limited context:

DB
DS
DW
ELSE
END
ENDIF
ENDM
EOT
EQU
EXITM
IF
IRP
IRPC
LOCAL
MACRO
ORG
REPT
SET

User-defined symbols are symbols you create to reference instruction addresses and data. These symbols are defined when they appear in the label field of an instruction or in the name field of EQU, SET, or MACRO assembler directives (see Chapters 5 and 6). Values for these symbols are determined modulo 64K although specific environments may limit the value even further. (See the subsection 'Permissible Range of Numbers,' earlier in this chapter.) Values outside these ranges cause an error.

Assembler-generated symbols are created by the assembler to replace user-defined symbols that have limited scope (limited to a macro definition).

NOTE

> Only instructions that allow registers as operands may have register-type operands. Expressions containing register-type operands are flagged as errors. The only assembler directives that may contain register-type operands are EQU, SET, and actual parameters in macro calls. Registers can be assigned alternate names only by EQU or SET.

Global and Limited Symbols

Symbols appearing as dummy parameters in a macro definition have limited scope and may only be used within that macro definition. Other symbols appearing in the body of a macro definition can be specified to have limited scope using the LOCAL assembler directive.

All other symbols, including macro definition *names,* have global scope and can be referenced from any part of your program. However, nested macro names cannot be called until all higher-level nested definitions have been called.

Permanent and Redefinable Symbols

Most symbols are permanent, that is, their values cannot change during the assembly operation. Only symbols defined with the SET and MACRO assembler directives are redefinable.

Duplicate Symbols

Local symbol names can be the same as reserved symbols, or local symbol names in other macro definitions. The assembler assigns a unique name to each local symbol.

A macro body containing a global label can be called only once. Additional calls cause 'multiply-defined symbol' errors. Attempts to redefine local or global symbols (other than with the SET directive) cause the same error.

3. MCS -48 ASSEMBLY LANGUAGE INSTRUCTIONS

This chapter describes the instruction set for the 8048, 8748, 8035, 8049, 8039, and 8021 (MCS-48) microcomputers. The 8041 and 8741 (UPI-41) microcomputers use essentially the same instructions. The few differences are described in Chapter 4.

The instructions are described here in four main functional groupings:

- Data Transfer
  - Within memory
  - Input/Output
- Data Manipulation
  - Logical operations
  - Bit rotation (shift)
  - Arithmetic
  - Miscellaneous accumulator operations
- Setting Program Controls
  - Condition bits, flags
  - Timer/event counter
  - Interrupts
  - Register and memory banks
  - NOP
- Transferring Program Control
  - Subroutine call
  - Return from subroutine
  - Jump operations

Most MCS-48 instructions require one machine cycle for execution (2.5 microseconds for the 8048, 10 microseconds for the 8021). Exceptions are I/O instructions, instructions using immediate data, subroutine calls and returns, jumps, and certain data transfers within memory, which require two cycles.

NOTE

For microcomputers having more than 1K of program memory, the only instructions that can reside in the last byte of a 2K block (locations 2047, 4095) are the subroutine returns (RET, RETR) and the second byte of a jump instruction. Exceptions cause a displacement (D) error. See Appendix F.

DATA TRANSFER INSTRUCTIONS

Data Transfer Within 8048 Memory

This section describes those instructions used to move data within resident and external 8048 data memory and program memory. This includes the MOV, MOVX, and MOVP data move instructions and the XCH and SWAP data exchange instructions. The move instructions overlay existing data in the target location. Data in the source location is unchanged. The exchange instructions swap data between two locations.

Register/Accumulator Moves

Data can be transferred between 8048 data memory working registers 0-7 and the accumulator by addressing the registers directly (R0-R7). R0-R7 can refer to data memory locations 0-7 if register bank 0 has been selected or to locations 24-31 if register bank 1 has been selected. Register bank 0 is the initialization value.

Move Register Contents to Accumulator

Op Code	*Operands*	
MOV	A,Rr	r=0-7

1 1 1 1	1 r r r

Eight bits of data are moved from working register 'r' into the accumulator.

Example:

```
MAR:   MOV    A,R3     ;MOVE CONTENTS OF REG
                       ;3 TO ACC
```

Move Accumulator Contents to Register

Op Code	*Operands*	
MOV	Rr,A	r=0-7

1 0 1 0	1 r r r

The contents of the accumulator are moved to register 'r'.

Example:

```
MRA:   MOV    R0,A     ;MOVE CONTENTS OF
                       ;ACC TO REG 0
```

Data-Memory/Accumulator Moves

Data moves between the accumulator and nonregister locations in data memory are accomplished by placing the address of the memory location in either register 0 or register 1 of the currently selected register bank. This is called indirect addressing. The assembler knows that indirect addressing is intended by the 'commercial at' sign (@) preceding the register reference.

The MOV instructions reference locations 0-63 (8048) or locations 0-127 (8049) in resident data memory. The MOVX instructions reference locations 0-255 in the optional external data memory.

Move Data Memory Contents to Accumulator

Opcode	*Operands*	
MOV	A,@Rr	r=0-1

1 1 1 1	0 0 0 r

The contents of the data memory location addressed by bits 0-5 (8048) or bits 0-6 (8049) of register 'r' are moved to the accumulator. Register 'r' contents are unaffected.

Example: Assume R1 contains 00110110.

```
MADM:    MOV    A,@R1      ;MOVE CONTENTS OF DATA MEM
                           ;LOCATION 54 TO ACC
```

Move Accumulator Contents to Data Memory

Opcode	*Operands*	
MOV	@Rr,A	r=0-1

1 0 1 0	0 0 0 r

The contents of the accumulator are moved to the data memory location whose address is specified by bits 0-5 (8048) or bits 0-6 (8049) of register 'r'. Register 'r' contents are unaffected.

Example: Assume R0 contains 00000111.

```
MDMA:    MOV    @R0,A      ;MOVE CONTENTS OF ACC
                           ;TO LOCATION 7 (REG 7)
```

Move External-Data-Memory Contents to Accumulator

Opcode	*Operands*	
MOVX	A,@Rr	r=0-1

1 0 0 0	0 0 0 r

This is a 2-cycle instruction. The contents of the external data memory location addressed by register 'r' are moved to the accumulator. Register 'r' contents are unaffected. This instruction is not recognized by the 8021.

Example: Assume R1 contains 01110110.

```
MAXDM:      MOVX  A,@R1  ;MOVE CONTENTS OF
                         ;LOCATION 118 TO ACC
```

Move Accumulator Contents to External Data Memory

Opcode	*Operands*	
MOVX	@Rr,A	r=0-1

1 0 0 1	0 0 0 r

This is a 2-cycle instruction. The contents of the accumulator are moved to the external data memory location addressed by register 'r'. Register 'r' contents are unaffected. This instruction is not recognized by the 8021.

Example: Assume R0 contains 11000111.

```
MXDMA:      MOVX  @R0,A  ;MOVE CONTENTS OF ACC TO
                         ;LOCATION 199 IN EXTERNAL
                         ;DATA MEMORY
```

Immediate-Data Moves

Data can be inserted directly into the accumulator, a working register, or resident data memory using the move-immediate-data instructions. Immediate data can be in the form of an ASCII constant, a number, an expression to be evaluated at assembly time, a symbol name, or an instruction enclosed in parentheses. (See Chapter 2, the subsection 'Operand Field.') The assembler recognizes immediate data by the 'pound sign' (#) preceding such data.

Immediate data must evaluate to a number that can be expressed in eight bits (that is, less than 256 decimal). Larger numbers generate an error condition. Larger numbers can be placed in data memory, however, by moving immediate data to adjoining locations.

Move Immediate Data to Register

Opcode	*Operands*	
MOV	**Rr,#data**	r=0-7

1 0 1 1	1 r r r	data

This is a 2-cycle instruction. The 8-bit value specified by 'data' is moved to register 'r.'

Examples:

```
MIR4:  MOV   R4,#HEXTEN    ;THE VALUE OF THE SYMBOL
                           ;'HEXTEN' IS MOVED INTO
                           ;REG 4
MIR5:  MOV   R5,#PI*(R*R)  ;THE VALUE OF THE
                           ;EXPRESSION 'PI*(R*R) IS
                           ;MOVED INTO REG 5
MIR6:  MOV   R6,#0ACH      ;'AC' HEX IS MOVED INTO
                           ;REG 6
```

Move Immediate Data to Data Memory

Opcode	*Operands*	
MOV	**@Rr,#data**	r=0-1

1 0 1 1	0 0 0 r	data

This is a 2-cycle instruction. The 8-bit value specified by 'data' is moved to the resident data memory location addressed by register 'r,' bits 0-5 (8048) or bits 0-6 (8049).

Example: Move the hexadecimal value AC3F to locations 62-63.

```
MIDM:  MOV   R0,#62        ;MOVE '62' DEC TO ADDR REG 0
       MOV   @R0,#0ACH     ;MOVE 'AC' HEX TO LOCATION 62
       INC   R0            ;INCREMENT REG 0 TO '63'
       MOV   @R0,#3FH      ;MOVE '3F' HEX TO LOCATION 63
```

Move Immediate Data to Accumulator

Opcode	*Operands*
MOV	**A,#data**

0 0 1 0	0 0 1 1	data

This is a 2-cycle instruction. The 8-bit value specified by 'data' is moved to the accumulator.

Example:

```
        MOV    A,#0A3H           ;MOVE 'A3' HEX TO ACC
```

PSW/Accumulator Moves

Data can be moved back and forth between the program status word and the accumulator. This is particularly useful for manipulating the stack pointer (PSW bits 0-2), which cannot be altered by specific instruction (as can the carry, flag 0, and register bank switch bits in the PSW).

Move PSW Contents to Accumulator

Opcode	*Operands*
MOV	**A,PSW**

1 1 0 0	0 1 1 1

The contents of the program status word are moved to the accumulator. This instruction is not recognized by the 8021.

Example: Jump to 'RB1SET' routine if PSW bank switch, bit 4, is set.

```
        BSCHK:  MOV     A,PSW       ;MOVE PSW CONTENTS TO ACC
                JB4     RB1SET      ;JUMP TO 'RB1SET' IF ACC
                                    ;BIT 4=1
```

Move Accumulator Contents to PSW

Opcode	*Operands*
MOV	**PSW,A**

1 1 0 1	0 1 1 1

The contents of the accumulator are moved into the program status word. All condition bits and the stack pointer are affected by this move. This instruction is not recognized by the 8021.

Example: Move up stack pointer by two memory locations, that is, increment the pointer by one.

```
        INCPTR: MOV     A,PSW       ;MOVE PSW CONTENTS TO ACC
                INC     A           ;INCREMENT ACC BY ONE
                MOV     PSW,A       ;MOVE ACC CONTENTS TO PSW
```

Timer/Accumulator Moves

Data can be moved between the accumulator and the special timer/event-counter register. This allows initialization and monitoring of this register's contents.

Move Timer/Counter Contents to Accumulator

Opcode	*Operands*
MOV	A,T

0 1 0 0	0 0 1 0

The contents of the timer/event-counter register are moved to the accumulator.

Example: Jump to 'EXIT' routine when timer reaches '64,' that is, when bit 6 is set – assuming initialization <64.

```
TIMCHK:  MOV    A,T          ;MOVE TIMER CONTENTS TO ACC
         JB6    EXIT         ;JUMP TO 'EXIT' IF ACC BIT 6=1
```

Move Accumulator Contents to Timer/Counter

Opcode	*Operands*
MOV	T,A

0 1 1 0	0 0 1 0

The contents of the accumulator are moved to the timer/event-counter register.

Example: Initialize and start event counter.

```
INITEC:  CLR    A            ;CLEAR ACC TO ZEROS
         MOV    T,A          ;MOVE ZEROS TO EVENT COUNTER
         STRT   CNT          ;START COUNTER
```

Program-Memory/Accumulator Moves

Data in program memory can be accessed indirectly using the accumulator as an address register. The accumulator reference is preceded by a 'commercial at' sign (@) to indicate indirection. The 8-bit address in the accumulator is used to reference a location in program memory; the contents of the memory location are then moved to the accumulator.

The 8-bit address limits the range of a program memory reference to the current 256-location page. One special instruction allows you to reference page 3 (locations 768-1023) from any location in program memory, however. This convenience lets you group frequently-accessed information (such as tables or indexes) in one easily-read area.

Move Current Page Data to Accumulator

Opcode	*Operands*
MOVP	A,@A

1 0 1 0	0 0 1 1

The contents of the program memory location addressed by the accumulator are moved to the accumulator. Only bits 0-7 of the program counter are affected, limiting the program memory reference to the current page. The program counter is restored following this operation.

NOTE

This is a 1-byte, 2-cycle instruction. If it appears in location 255 of a program memory page, @A addresses a location in the *following* page.

Example:

```
MOV128: MOV     A,#128     ;MOVE '128' DEC TO ACC
        MOVP    A,@A       ;CONTENTS OF 129TH LOCATION
                           ;IN CURRENT PAGE ARE MOVED
                           ;TO ACC
```

Move Page 3 Data to Accumulator

Opcode	*Operands*
MOVP3	A,@A

1 1 1 0	0 0 1 1

This is a 2-cycle instruction. The contents of the program memory location (within page 3) addressed by the accumulator are moved to the accumulator. The program counter is restored following this operation. This instruction is not recognized by the 8021.

Example: Look up ASCII equivalent of hexadecimal code in table contained at the beginning of page 3. Note that ASCII characters are designated by a 7-bit code; the eighth bit is always reset (see Appendix E).

```
TABSCH: MOV     A,#0B8H    ;MOVE 'B8' HEX TO ACC (10111000)
        ANL     A,#7FH     ;LOGICAL AND ACC TO MASK BIT 7
                           ;(00111000)
        MOVP3   A,@A       ;MOVE CONTENTS OF LOCATION '38'
                           ;HEX IN PAGE 3 TO ACC (ASCII '8')
```

Example: Access contents of location in page 3 labeled TAB1. Assume current program location is not in page 3.
NOTE: The LOW operator is described in Chapter 2, 'Assembly-Time Expression Evaluation.'

```
TABSCH:  MOV     A,#LOW TAB1     ;ISOLATE BITS 0-7 OF LABEL
                                 ;ADDRESS VALUE
         MOVP3   A,@A            ;MOVE CONTENTS OF PAGE 3 LOCATION
                                 ;LABELED 'TAB1' TO ACC
```

Data Exchange Instructions

Data can be exchanged between the accumulator and working registers specified directly, or between the accumulator and data memory locations specified indirectly (preceded by @). The exchange instructions apply only to resident data memory, and not to external memory.

The main advantage of a data exchange over a simple move is that data at the target location is not lost and can be moved back to its original location if necessary. Binary Coded Decimal (BCD) arithmetic can be performed on the 8048 by dividing 8-bit values into two 4-bit BCD digits. Two instructions, XCHD and SWAP, allow transfer of such 4-bit digits.

Exchange Accumulator-Register Contents

Opcode	*Operands*	
XCH	A,Rr	r=0-7

0 0 1 0	1 r r r

The contents of the accumulator and the contents of working register 'r' are exchanged.

Example:

Move PSW contents to Reg 7 without losing accumulator contents.

```
XCHAR7:   XCH    A,R7      ;EXCHANGE CONTENTS OF REG 7
                           ;AND ACC
          MOV    A,PSW     ;MOVE PSW CONTENTS TO ACC
          XCH    A,R7      ;EXCHANGE CONTENTS OF REG 7
                           ;AND ACC AGAIN
```

Exchange Accumulator and Data Memory Contents

Opcode	*Operands*	
XCH	A,@Rr	r=0-1

0 0 1 0	0 0 0 r

The contents of the accumulator and the contents of the data memory location addressed by bits 0-5 (8048) or bits 0-6 (8049) of register 'r' are exchanged. Register 'r' contents are unaffected.

Example: Decrement contents of location 52.

```
DEC52:   MOV    R0,#52        ;MOVE '52' DEC TO ADDRESS
                              ;REG 0
         XCH    A,@R0         ;EXCHANGE CONTENTS OF ACC
                              ;AND LOCATION 52
         DEC    A             ;DECREMENT ACC CONTENTS
         XCH    A,@R0         ;EXCHANGE CONTENTS OF ACC
                              ;AND LOCATION 52 AGAIN
```

Exchange Accumulator and Data Memory 4-Bit Data

Opcode	*Operands*	
XCHD	**A,@Rr**	r=0-1

0	0	1	1	0	0	0	r

This instruction exchanges bits 0-3 of the accumulator with bits 0-3 of the data memory location addressed by bits 0-5 (8048) or bits 0-6 (8049) of register 'r.' Bits 4-7 of the accumulator, bits 4-7 of the data memory location, and the contents of register 'r' are unaffected.

Example: Assume program counter contents have been stacked in locations 22-23.

```
XCHNIB:  MOV     R0,#23    ;MOVE '23' DEC TO REG 0
         CLR     A         ;CLEAR ACC TO ZEROS
         XCHD    A,@R0     ;EXCHANGE BITS 0-3 OF ACC
                           ;AND LOCATION 23 (BITS
                           ;8-11 OF PC ARE ZEROED,
                           ;ADDRESS REFERS TO PAGE 0)
```

Swap 4-Bit Data Within Accumulator

Opcode	*Operand*
SWAP	**A**

0	1	0	0	0	1	1	1

Bits 0-3 of the accumulator are swapped with bits 4-7 of the accumulator.

Example: Pack bits 0-3 of locations 50-51 into location 50.

```
PCKDIG: MOV     R0,#50   ;MOVE '50' DEC TO REG 0
        MOV     R1,#51   ;MOVE '51' DEC TO REG 1
        XCHD    A,@R0    ;EXCHANGE BITS 0-3 OF ACC
                         ;AND LOCATION 50
        SWAP    A        ;SWAP BITS 0-3 AND 4-7 OF
                         ;ACC
        XCHD    A,@R1    ;EXCHANGE BITS 0-3 OF ACC
                         ;AND LOCATION 51
        MOV     @R0,A    ;MOVE CONTENTS OF ACC TO
                         ;LOCATION 50
```

Input/Output Data Transfers

The MCS-48 input/output instructions allow data to be transferred between the accumulator and I/O ports. As was described in Chapter 1 (the subsection 'Input/Output'), the BUS port and ports 0-2 are used for standard I/O operations. Ports 4-7 on the 8243 expander, consisting of four pins each, can be attached through port 2, pins P20-23, to provide 16 additional I/O lines. Port 0 is used only by the 8021, as it does not have a BUS port.

All input/output data transfers are 2-cycle operations.

Standard I/O Transfers

The BUS port and ports 0-2 can be either input or output ports depending on the instruction flow. The BUS port actually has two modes of operation. If the MCS-48 is used as a freestanding device, the BUS acts as a general I/O port like ports 0-2. If the MCS-48 is part of a more extensive system with expanded memory and I/O, the BUS is a bidirectional port with synchronous strobes. Bus lines are latched only for single-device (freestanding) operations.

Input Port 0-2 Data to Accumulator

Opcode	*Operands*	
IN	A,Pp	p=0-2

```
|0 0 0 0|1 0 p p|
```

Data present on port 'p' is transferred (read) to the accumulator. Port '0' can be specified only for the 8021.

Example:

```
INP12: IN    A,P1     ;INPUT PORT 1 CONTENTS TO ACC
       MOV   R6,A     ;MOVE ACC CONTENTS TO REG 6
       IN    A,P2     ;INPUT PORT 2 CONTENTS TO ACC
       MOV   R7,A     ;MOVE ACC CONTENTS TO REG 7
```

0FFH (ones) should be written to ports 1 and 2 before using them as inputs (using the OUTL Pp,A instruction described below.

Strobed Input of BUS Data to Accumulator

Opcode	*Operands*
INS	A,BUS

0 0 0 0	1 0 0 0

Data present on the BUS port is transferred (read) to the accumulator when the RD pulse is dropped. (Refer to timing diagrams in the MCS-48 user's manual for details.) This instruction is not recognized by the 8021.

Example:

```
INPBUS:   INS   A,BUS          ;INPUT BUS CONTENTS TO ACC
```

Output Accumulator Data to Port 0-2

Opcode	*Operands*	*Opcode*	*Operands*	
OUTL	P0, A	OUTL	Pp, A	p=1-2

1 0 0 1	0 0 0 0	0 0 1 1	1 0 p p

Data residing in the accumulator is transferred (written) to port 'p' and latched. Port '0' can be specified only for the 8021.

Example:

```
OUTLP:   MOV    A,R7     ;MOVE REG 7 CONTENTS TO ACC
         OUTL   P2,A     ;OUTPUT ACC CONTENTS TO PORT 2
         MOV    A,R6     ;MOVE REG 6 CONTENTS TO ACC
         OUTL   P1,A     ;OUTPUT ACC CONTENTS TO PORT 1
```

Output Accumulator Data to BUS

Opcode	*Operands*
OUTL	BUS,A

0 0 0 0	0 0 1 0

Data residing in the accumulator is transferred (written) to the BUS port and latched. The latched data remains valid until altered by another OUTL instruction. Any other instruction requiring use of the BUS port (except INS) destroys the contents of the BUS latch. This includes expanded memory operation (such as the MOVX instruction).

Logical operations on BUS data (AND and OR) assume the OUTL BUS,A instruction has been issued previously. This instruction is not recognized by the 8021.
Example:

```
OUTLBP:  OUTL   BUS,A        ;OUTPUT ACC CONTENTS TO BUS
```

Expanded I/O Transfers

Data can be transferred between the accumulator and ports 4-7 on the 8243 expander device using the MOVD instructions. The 8243 attaches to port 2 pins P20-23 and existing P20-23 data is destroyed by these instructions.

Ports 4-7 are four pins each. The MOVD instructions transfer data to/from bits 0-3 of the accumulator.

Move Port 4-7 Data to Accumulator

Opcode	*Operands*	
MOVD	A,Pp	p=4-7

0	0	0	0	1	1	p	p

Data on 8243 port 'p' is moved (read) to accumulator bits 0-3. Accumulator bits 4-7 are zeroed.

NOTE

Bits 0-1 of the opcode are used to represent ports 4-7. If you are coding in binary rather than assembly language, the mapping is as follows:

Bits 1 0	*Port*
0 0	4
0 1	5
1 0	6
1 1	7

Example:

```
INPPT5:   MOVD  A,P5        ;MOVE PORT 5 DATA TO ACC
                            ;BITS 0-3, ZERO ACC BITS
                            ;4-7
```

Move Accumulator Data to Port 4, 5, 6, or 7

Opcode	*Operands*	
MOVD	Pp,A	p=4-7

0	0	1	1	1	1	p	p

Data in accumulator bits 0-3 is moved (written) to 8243 port 'p.' Accumulator bits 4-7 are unaffected. (See NOTE above regarding port mapping.)

Example: Move data in accumulator to ports 4 and 5.

```
OUTP45:  MOVD    P4,A     ;MOVE ACC BITS 0-3 TO PORT 4
         SWAP    A        ;EXCHANGE ACC BITS 0-3 AND
                          ;4-7
         MOVD    P5,A     ;MOVE ACC BITS 0-3 TO PORT 5
```

DATA MANIPULATION INSTRUCTIONS

The MCS-48 instruction set includes 34 instructions for manipulating data including logical operations, bit rotation, incrementing and decrementing of data, addition, and miscellaneous accumulator operations.

Logical Operations

Operations in this category include logical AND, OR, and EXCLUSIVE OR (XOR). Assuming an initial value of 11100111, a mask of 10101010 would produce the following results following these operations.

```
     11100111          11100111          11100111
AND  10101010      OR  10101010     XOR  10101010
     --------          --------          --------
     10100010          11101111          01001101

  (=1 if both       (=1 if either      (=1 if bits
  are 1)            is 1)              different)
```

Most of the logical instructions operate on values in the accumulator. However the 8048 also allows logical AND and OR operations on data residing in I/O ports.

Accumulator/Register Logical Operations

In the following three instructions, the specified working register contains the mask to be combined logically with an accumulator value. The result of the operation remains in the accumulator.

Logical AND Accumulator With Register Mask

Opcode	*Operands*	
ANL	A,Rr	r=0-7

0	1	0	1	1	r	r	r

Data in the accumulator is logically ANDed with the mask contained in working register 'r.'

Example:

```
ANDREG: ANL    A,R3          ;'AND' ACC CONTENTS WITH
                             ;MASK IN REG 3
```

Logical OR Accumulator with Register Mask

Opcode	*Operands*	
ORL	A,Rr	r=0-7

0	1	0	0	1	r	r	r

Data in the accumulator is logically ORed with the mask contained in working register 'r.'

Example:

```
ORREG:  ORL    A,R4          ;'OR' ACC CONTENTS WITH
                             ;MASK IN REG 4
```

Logical XOR Accumulator With Register Mask

Opcode	*Operands*	
XRL	A,Rr	r=0-7

1	1	0	1	1	r	r	r

Data in the accumulator is EXCLUSIVE ORed with the mask contained in working register 'r.'

Example:

```
XORREG: XRL    A,R5          ;'XOR' ACC CONTENTS WITH MASK IN
                             ;REG 5
```

Accumulator/Data-Memory Logical Operations

The mask for a logical operation can reside anywhere in resident data memory. (Logical operations cannot reference external memory.) The address of the mask source is contained in Register 0 or Register 1. Indirect addressing is indicated by the '@' preceding the register reference.

The value to be masked and result reside in the accumulator.

Logical AND Accumulator With Memory Mask

Opcode	*Operands*	
ANL	A,@Rr	r=0-1

0	1	0	1	0	0	0	r

Data in the accumulator is logically ANDed with the mask contained in the data memory location referenced by register 'r,' bits 0-5 (8048) or bits 0-6 (8049).

Example:

```
ANDDM:  MOV    R0,#0FFH     ;MOVE 'FF' HEX TO REG 0
        ANL    A,@R0        ;'AND' ACC CONTENTS WITH MASK
                            ;IN LOCATION 63
```

Logical OR Accumulator With Memory Mask

Opcode	*Operands*	
ORL	A,@Rr	r=0-1

0	1	0	0	0	0	0	r

Data in the accumulator is logically ORed with the mask contained in the data memory location referenced by register 'r,' bits 0-5 (8048) or bits 0-6 (8049).

Example:

```
ORDM:  MOV    R0,#3FH       ;MOVE '3F' HEX TO REG 0
       ORL    A,@R0         ;'OR' ACC CONTENTS WITH MASK
                            ;IN LOCATION 63
```

Logical XOR Accumulator With Memory Mask

Opcode	*Operands*	
XRL	A,@Rr	r=0-1

1	1	0	1	0	0	0	r

Data in the accumulator is EXCLUSIVE ORed with the mask contained in the data memory location addressed by register 'r,' bits 0-5 (8048) or bits 0-6 (8049).

Example:

```
XORDM:  MOV    R1,#20H       ;MOVE '20' HEX TO REG 1
        XRL    A,@R1         ;'XOR' ACC CONTENTS WITH MASK
                             ;IN LOCATION 32
```

Accumulator/Immediate-Data Logical Operations

The mask to be combined logically with the accumulator value can be specified as 'immediate' data. This data is recognized by the preceding pound sign (#) and must evaluate to eight bits. All instructions specifying immediate data require two cycles for execution.

The result of the logical operation remains in the accumulator.

Logical AND Accumulator with Immediate Mask

Opcode	*Operands*
ANL	**A,#data**

0 1 0 1	0 0 1 1	data

Data in the accumulator is logically ANDed with an immediately-specified mask.

Examples:

```
ANDID: ANL    A,#0AFH       ;'AND' ACC CONTENTS WITH
                            ;MASK 10101111

       ANL    A,#3+X/Y      ;'AND' ACC CONTENTS WITH
                            ;VALUE OF EXP '3 + X/Y'
```

Logical OR Accumulator With Immediate Mask

Opcode	*Operands*
ORL	**A,#data**

0 1 0 0	0 0 1 1	data

Data in the accumulator is logically ORed with an immediately-specified mask.

Example:

```
ORID:  ORL    A,#'X'        ;'OR' ACC CONTENTS WITH MASK
                            ;01011000 (ASCII VALUE OF 'X')
```

Logical XOR Accumulator With Immediate Mask

Opcode	*Operands*
XRL	**A,#data**

1 1 0 1	0 0 1 1	data

Data in the accumulator is EXCLUSIVE ORed with an immediately-specified mask.

Example:

```
XORID:    XOR    A,#HEXTEN  ;XOR CONTENTS OF ACC WITH
                            ;MASK EQUAL VALUE OF
                            ;SYMBOL 'HEXTEN'
```

Input/Output Logical Operations

Data residing on the BUS port or ports 1 and 2 can be logically combined with an immediately-specified mask. The mask data must be preceded by '#' and must evaluate to eight bits. Data on 8243 ports 4-7 can be logically combined with a mask contained in bits 0-3 of the accumulator. In the case of the 8021, I/O logical operations are permitted on 8243 ports only.

Only AND and OR logical operations can be done on I/O data. XOR is not possible. The results of the logical operation remain on the specified port. All of the instructions described in this subsection require two cycles for execution. These instructions can be used to clear/set any specified outputs.

Logical AND Port 1-2 With Immediate Mask

Opcode	*Operands*	
ANL	**Pp,#data**	p=1-2

1 0 0 1	1 0 p p	data

Data on port 'p' is logically ANDed with an immediately-specified mask. This instruction is not recognized by the 8021.

Example:

```
ANDP2:    ANL    P2,#0F0H     ;'AND' PORT 2 CONTENTS WITH
                              ;MASK 'F0' HEX (CLEAR P20-23)
```

Logical AND BUS With Immediate Mask

Opcode	*Operands*
ANL	**BUS,#data**

1 0 0 1	1 0 0 0	data

Data on the BUS port is logically ANDed with an immediately-specified mask. This instruction assumes prior specification of an 'OUTL BUS,A' instruction. The 8021 does not recognize this instruction.

Example:

```
ANDBUS:  ANL   BUS,#MASK   ;'AND' BUS CONTENTS WITH
                           ;MASK EQUAL VALUE OF
                           ;SYMBOL 'MASK'
```

Logical OR Port 1-2 With Immediate Mask

Opcode	*Operands*	
ORL	**Pp,#data**	p=1-2

1 0 0 0	1 0 p p	data

Data on port 'p' is logically ORed with an immediately-specified mask. This instruction is not recognized by the 8021.

Example:

```
ORP1:  ORL   P1,#0FFH      ;'OR' PORT 1 CONTENTS WITH
                           ;MASK 'FF' HEX (SET PORT 1
                           ;TO ALL ONES)
```

Logical OR BUS With Immediate Mask

Opcode	*Operands*
ORL	**BUS,#data**

1 0 0 0	1 0 0 0	data

Data on the BUS port is logically ORed with an immediately-specified mask. This instruction assumes prior specification of an 'OUTL BUS,A' instruction. The 8021 does not recognize this instruction.

Example:

```
ORBUS:   ORL   BUS,#HEXMSK ;'OR' BUS CONTENTS WITH
                           ;MASK EQUAL VALUE OF
                           ;SYMBOL 'HEXMSK'
```

Logical AND Port 4-7 With Accumulator Mask

Opcode	*Operands*	
ANLD	**Pp,A**	p=4-7

1 0 0 1	1 1 p p

Data on port 'p' is logically ANDed with the digit mask contained in accumulator bits 0-3 and the result written to port 'p.' The accumulator is not affected.

NOTE

The mapping of port 'p' to opcode bits 0-1 is as follows:

1	*0*	*Port*
0	0	4
0	1	5
1	0	6
1	1	7

Example:

```
ANDP4:    ANLD    P4,A          ;'AND' PORT 4 CONTENTS WITH
                                ;ACC BITS 0-3
```

Logical OR Port 4-7 with Accumulator Mask

Opcode	*Operands*	
ORLD	Pp,A	p=4-7

1	0	0	0	1	1	p	p

Data on port 'p' is logically ORed with the digit mask contained in accumulator bits 0-3 and the result is written to port 'p.' The accumulator is not affected. (See the NOTE accompanying the preceding instruction.)

Example:

```
ORP7:  ORLD   P7,A           ;'OR' PORT 7 CONTENTS
                             ;WITH ACC BITS 0-3
```

Rotate Operations

The MCS-48 instruction set includes four instructions for bit rotation of accumulator contents: right and left rotations that do not affect the carry bit, and rotations through the carry. All four instructions perform 'wraparound' rotations, as shown in Figure 3-1.

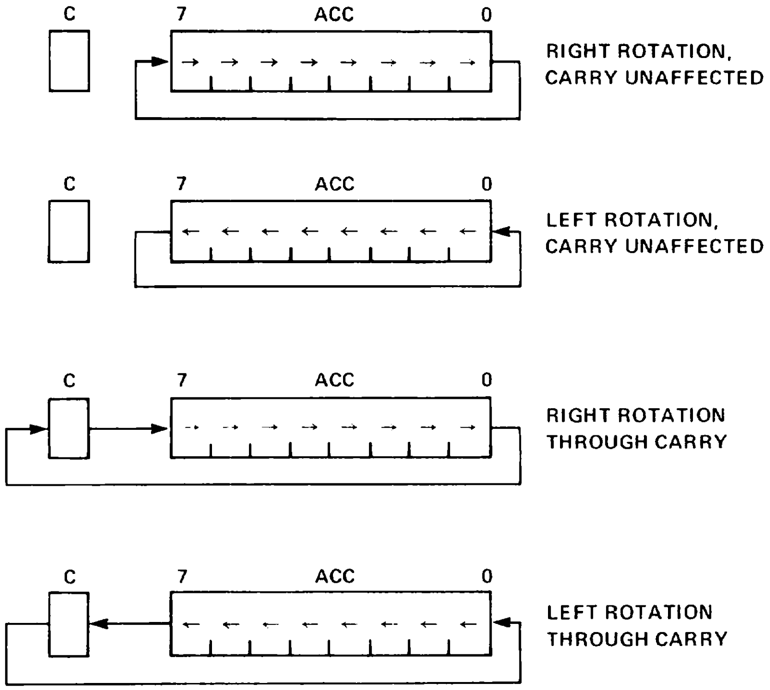


Figure 3-1 Bit Rotation

Rotate Without Carry

Rotate Right Without Carry

Opcode	*Operand*
RR	A

0	1	1	1	0	1	1	1

The contents of the accumulator are rotated right one bit. Bit 0 is rotated into the bit 7 position (Figure 3-1).

Example: Assume accumulator contains 10110001.

```
RRNC:    RR    A    ;NEW ACC CONTENTS ARE 11011000
```

Rotate Left Without Carry

Opcode	*Operand*
RL	A

1	1	1	0	0	1	1	1

The contents of the accumulator are rotated left one bit. Bit 7 is rotated into the bit 0 position (Figure 3-1).

Example: Assume accumulator contains 10110001.

```
RLNC:  RL    A     ;NEW ACC CONTENTS ARE 01100011
```

Rotate Through Carry

Rotate Right Through Carry

Opcode	*Operand*
RRC	A

0	1	1	0	0	1	1	1

The contents of the accumulator are rotated right one bit. Bit 0 replaces the carry bit; the carry bit is rotated into the bit 7 position (Figure 3-1).

Example: Assume carry is not set and accumulator contains 10110001.

```
RRTC:  RRC    A     ;CARRY IS SET AND ACC
                    ;CONTAINS 01011000
```

Rotate Left Through Carry

Opcode	*Operand*
RLC	A

1	1	1	1	0	1	1	1

The contents of the accumulator are rotated left one bit. Bit 7 replaces the carry bit; the carry bit is rotated into the bit 0 position (Figure 3-1).

Example: Assume accumulator contains a ‘signed’ number; isolate sign without changing value.

```
RLTC:  CLR  C   ;CLEAR CARRY TO ZERO
       RLC  A   ;ROTATE ACC LEFT;SIGN
                ;BIT (7) IS PLACED IN CARRY
       RR   A   ;ROTATE ACC RIGHT – VALUE
                ;(BITS 0-6) IS RESTORED, CARRY
                ;UNCHANGED, BIT 7 IS ZERO
```

Arithmetic Operations

Arithmetic operations include the increment, decrement, and addition instructions.

Increment/Decrement Instructions

You can increment (by one) the contents of the accumulator, a working register, or resident data memory location. The accumulator and working registers can be decremented. (External data memory contents cannot be incremented or decremented directly, although such data can be manipulated in the accumulator.)

The DJNZ instruction allows you to decrement a register, test for zero, and transfer program control accordingly. The register can be used as a counter, providing program loop control.

Increment Accumulator

Opcode	*Operand*
INC	**A**

0 0 0 1	0 1 1 1

The contents of the accumulator are incremented by one.

Example: Increment contents of location 100 in external data memory.

```
INCA:  MOV    R0,#100       ;MOVE '100' DEC TO
                            ;ADDRESS REG 0
       MOVX   A,@R0         ;MOVE CONTENTS OF LOCATION
                            ;100 TO ACC
       INC    A             ;INCREMENT A
       MOVX   @R0,A         ;MOVE ACC CONTENTS TO
                            ;LOCATION 100
```

Increment Register

Opcode	*Operand*	
INC	**Rr**	r=0-7

0 0 0 1	1 r r r

The contents of working register 'r' are incremented by one.

Example:

```
INCR0:   INC  R0          ;INCREMENT ADDRESS REG 0
```

Increment Data Memory Location

Opcode	*Operand*	
INC	**@Rr**	r=1-2

0 0 0 1	0 0 0 r

The contents of the resident data memory location addressed by register 'r' bits 0-5 (8048) or bits 0-6 (8049) are incremented by one.

Example:

```
INCDM:   MOV   R1,#3FH       ;MOVE ONES TO BITS 0-5
         INC   @R1           ;INCREMENT LOCATION 63
```

Decrement Accumulator

Opcode	*Operand*
DEC	**A**

0 0 0 0	0 1 1 1

The contents of the accumulator are decremented by one.

Example: Decrement contents of external data memory location 63.

```
MOV    R0,#3FH           ;MOVE '3F' HEX TO REG 0
MOVX   A,@R0             ;MOVE CONTENTS OF LOCATION
                         ;63 TO ACC
DEC    A                 ;DECREMENT ACC
MOVX   @R0,A             ;MOVE CONTENTS OF ACC TO LOCATION
                         ;63 IN EXPANDED MEMORY
```

Decrement Register

Opcode	*Operand*	
DEC	**Rr**	r=0-7

1 1 0 0	1 r r r

The contents of working register 'r' are decremented by one. This instruction is not recognized by the 8021.

Example:

```
DECR1:   DEC  R1             ;DECREMENT ADDRESS REG 1
```

Decrement Register and Test

Opcode	*Operand*	
DJNZ	Rr,address	r=0-7

1	1	1	0	1	r	r	r	address

This is a 2-cycle instruction. Register 'r' is decremented and tested for zero. If the register contains all zeros, program control falls through to the next instruction. If the register contents are not zero, control jumps to the specified 'address.'

The address in this case must evaluate to eight bits, that is, the jump must be to a location within the current 256-location page.

NOTE

> A 12-bit address specification does not cause an error if the DJNZ instruction and the jump target are on the same page. If the DJNZ instruction begins in location 255 of a page, it must jump to a target address on the following page.

Example: Increment values in data memory locations 50-54.

```
        MOV    R0,#50    ;MOVE '50' DEC TO ADDRESS REG 0
        MOV    R3,#5     ;MOVE '5' DEC TO COUNTER REG 3
INCRT:  INC    @R0       ;INCREMENT CONTENTS OF LOCATION
                         ;ADDRESSED BY REG 0
        INC    R0        ;INCREMENT ADDRESS IN REG 0
        DJNZ   R3,INCRT  ;DECREMENT REG 3 – JUMP TO
                         ;'INCRT' IF REG 3 NONZERO
        NEXT   ---       ;'NEXT' ROUTINE EXECUTED IF
                         ;R3 IS ZERO
```

Addition Instructions

The contents of working registers or other resident data-memory locations, or immediately-specified data, can be added to the contents of the accumulator. The result remains in the accumulator.

As described earlier, data memory locations are addressed indirectly through registers 0-1. The reference to these registers is preceded by '@' to indicate indirection. Immediately-specified data is preceded by '#' and must evaluate to eight bits. All immediate operations require two cycles for execution.

Addition can be performed 'with carry.' This means that the value in the carry bit is added to the accumulator at the low-order end and the carry bit is set to zero automatically before the regular addition operation takes place. This is necessary, for example, when adding 16-bit values, to ensure that any carry from the low-order byte addition is reflected in the high-order byte addition.

Example: Add value 10101010 to accumulator value 10000010 with carry. Assume carry bit is currently set.

STEP 1: ADD C to ACC and zero C

C	7				ACC			0
0	1	0	1	0	1	0	1	1

STEP 2: ADD 10000010 to ACC; overflow to C if necessary

C	7				ACC			0
1	0	0	1	0	1	1	0	1

All addition operations (with or without carry) affect the carry and auxiliary carry bits in the event of an addition overflow.

Add Register Contents to Accumulator

Opcode	*Operands*	
ADD	**A,Rr**	r=0-7

0	1	1	0	1	r	r	r

The contents of register 'r' are added to the accumulator.

Example:

```
ADDREG:    ADD    A,R6      ;ADD REG 6 CONTENTS TO ACC
```

Add Carry and Register Contents to Accumulator

Opcode	*Operands*	
ADDC	**A,Rr**	r=0-7

0	1	1	1	1	r	r	r

The content of the carry bit is added to accumulator bit 0 and the carry bit cleared. The contents of register 'r' are then added to the accumulator.

Example:

```
ADDRGC:    ADDC   A,R4      ;ADD CARRY AND REG 4 CONTENTS
                            ;TO ACC
```

Add Data Memory Contents to Accumulator

Opcode	*Operands*	
ADD	**A,@Rr**	r=0-1

0 1 1 0	0 0 0 r

The contents of the standard data memory location addressed by register 'r' bits 0-5 (8048) or bits 0-6 (8049) are added to the accumulator.

Example:

```
ADDM:   MOV   R0,#2FH     ;MOVE '2F' HEX TO REG 0
        ADD   A,@R0       ;ADD VALUE OF LOCATION 47 TO
                          ;ACC
```

Add Carry and Data Memory Contents to Accumulator

Opcode	*Operands*	
ADDC	**A,@Rr**	r=0-1

0 1 1 1	0 0 0 r

The content of the carry bit is added to accumulator bit 0 and the carry bit is cleared. Then the contents of the standard data memory location addressed by register 'r' bits 0-5 (8048) or bits 0-6 (8049) are added to the accumulator.

Example:

```
ADDMC:  MOV    R1,#40     ;MOVE '40' DEC TO REG 1
        ADDC   A,@R1      ;ADD CARRY AND LOCATION 40
                          ;CONTENTS TO ACC
```

Add Immediate Data to Accumulator

Opcode	*Operands*
ADD	**A,#data**

0 0 0 0	0 0 1 1	data

This is a 2-cycle instruction. The specified data is added to the accumulator.

Example:

```
ADDID:  ADD   A,#ADDER   ;ADD VALUE OF SYMBOL
                         ;'ADDER' TO ACC
```

Add Carry and Immediate Data to Accumulator

Opcode	*Operands*
ADDC	A,#data

0	0	0	1	0	0	1	1	data

This is a 2-cycle instruction. The content of the carry bit is added to accumulator location 0 and the carry bit cleared. Then the specified data is added to the accumulator.

Example:

```
ADDIDC:  ADDC   A,#225        ;ADD CARRY AND '225'
                              ;DEC TO ACC
```

Miscellaneous Accumulator Operations

Three data manipulation instructions allow the accumulator contents to be cleared, complemented, or divided into two decimal digits.

Clear Accumulator

Opcode	*Operand*
CLR	A

0	0	1	0	0	1	1	1

The contents of the accumulator are cleared to zero.

Complement Accumulator

Opcode	*Operand*
CPL	A

0	0	1	1	0	1	1	1

The contents of the accumulator are complemented. This is strictly a one's complement. Each one is changed to zero and vice-versa. (See the discussion of arithmetic notation in Chapter 2, the subsection 'Two's Complement Arithmetic.')

Example: Assume accumulator contains 01101010.

```
CPLA:  CPL   A                ;ACC CONTENTS ARE
                              ;COMPLEMENTED TO 10010101
```

Decimal Adjust Accumulator

Opcode	*Operand*
DA	A

```
|0 1 0 1|0 1 1 1|
```

The 8-bit accumulator value is adjusted to form two 4-bit Binary Coded Decimal (BCD) digits (basically following an addition operation). The carry bit is affected.

If the contents of bits 0-3 are greater than nine, or if the auxiliary carry bit is one, the accumulator is incremented by six.

The four high-order bits are then checked. If bits 4-7 exceed nine, or if C is one, these bits are increased by six. If an overflow occurs, C is set to one; otherwise, it is cleared to zero.

Example: Assume accumulator contains 10011011.

```
DA   A     ;ACC ADJUSTED TO 00000001 WITH C SET

C AC   7     4    3     0
|0|0|  |1 0 0 1|  |1 0 1 1|
                   0 1 1 0    ADD SIX TO BITS 0-5
|0|0|  |1 0 1 0|  |0 0 0 1|
        0 1 1 0               ADD SIX TO BITS 4-7
|1|0|  |0 0 0 0|  |0 0 0 1|   OVERFLOW TO C
```

SETTING PROGRAM CONTROLS

Your program can be controlled by the setting of the condition bits, flags, and switches described in Chapter 1, the section 'Programmable Controls.' This section describes the instructions for manipulating these controls. It also describes interrupt controls, timer/event-counter controls, clock control, the selection of memory and register banks, and the NOP instruction.

Carry and Flag Controls

The carry bit (C), flag 0 (F0), and flag 1 (F1) can be cleared or complemented by the following instructions. Carry (PSW bit 7) and flag 0 (PSW bit 5) can also be manipulated by moving the PSW to the accumulator, masking the entire eight bits, then moving the result back to the PSW. This might be a preferable approach if several other bits in the PSW were being altered at the same time.

Clear Carry Bit

Opcode	*Operand*
CLR	C

```
|1 0 0 1|0 1 1 1|
```

During normal program execution, the carry bit can be set to one by the ADD, ADDC, RLC, RRC, CPL C, and DA instructions. This instruction resets the carry bit to zero.

Complement Carry Bit

Opcode	*Operand*
CPL	C

1 0 1 0	0 1 1 1

The setting of the carry bit is complemented; one is changed to zero, and zero is changed to one.

Example: Set C to one; current setting is unknown.

```
        CT01:  CLR    C                    ;C IS CLEARED TO ZERO
               CPL    C                    ;C IS SET TO ONE
```

Clear Flag 0

Opcode	*Operand*
CLR	F0

1 0 0 0	0 1 0 1

Flag 0 is cleared to zero. The 8021 does not recognize this instruction.

Complement Flag 0

Opcode	*Operand*
CPL	F0

1 0 0 1	0 1 0 1

The setting of flag 0 is complemented; one is changed to zero, and zero is changed to one. The 8021 does not recognize this instruction.

Clear Flag 1

Opcode	*Operand*
CLR	F1

1 0 1 0	0 1 0 1

Flag 1 is cleared to zero. The 8021 does not recognize this instruction.

Complement Flag 1

Opcode	*Operand*
CPL	F1

1	0	1	1	0	1	0	1

The setting of flag 1 is complemented; one is changed to zero, and zero is changed to one. The 8021 does not recognize this instruction.

Interrupt Controls

As described in Chapter 1, the 8048 responds to two kinds of interrupts: an 'external' interrupt initiated by a low signal on the interrupt input pin, and an overflow in the timer/event-counter register. The following instructions allow you to enable and disable these interrupts.

These interrupts and related instructions are not available on the 8021.

External Interrupt Control

If the external interrupt is enabled and the interrupt input pin goes to level zero, the interrupt sequence is activated. Control passes to program memory location 3, the program counter and bits 4-7 of the PSW are stored in the program stack, and the stack pointer (PSW bits 0-2) is incremented by one.

Enable External Interrupt

Opcode	*Operand*
EN	I

0	0	0	0	0	1	0	1

External interrupts are enabled. A low signal on the interrupt input pin initiates the interrupt sequence. This instruction is not recognized by the 8021.

Disable External Interrupt

Opcode	*Operand*
DIS	I

0	0	0	1	0	1	0	1

External interrupts are disabled. A low signal on the interrupt input pin has no effect. This instruction is not recognized by the 8021.

Timer/Counter Interrupt Control

If this interrupt is enabled and the timer/event-counter overflows, the interrupt sequence is activated. Control passes to program memory location 7, the program counter and PSW bits 4-7 are stored in the program stack, and the stack pointer incremented.

The timer flag (TF) is set when the timer/counter overflows, whether or not the interrupt is enabled. The timer continues to accumulate time after an overflow occurs.

Enable Timer/Counter Interrupt

Opcode	*Operand*
EN	TCNTI

0	0	1	0	0	1	0	1

Timer/counter interrupts are enabled. An overflow of this register initiates the interrupt sequence. This instruction is not recognized by the 8021.

Disable Timer/Counter Interrupt

Opcode	*Operand*
DIS	TCNTI

0	0	1	1	0	1	0	1

Timer/counter interrupts are disabled. Any pending timer interrupt request is cleared. The interrupt sequence is not initiated by an overflow, but the timer flag is set and time accumulation continues. This instruction is not recognized by the 8021.

Timer/Event-Counter Controls

The following instructions are used to start and stop time accumulation or event counting in the timer/event-counter register.

Start Timer

Opcode	*Operand*
STRT	T

0	1	0	1	0	1	0	1

Timer accumulation is initiated in the timer register. The register is incremented every 32 instruction cycles. The prescaler (where the 32 cycles are counted) is cleared, but the timer register is not.

Example: Initialize and start timer.

```
STARTT: CLR    A          ;CLEAR ACC TO ZEROS
        MOV    T,A        ;MOVE ZEROS TO TIMER
        EN     TCNTI      ;ENABLE TIMER INTERRUPT
        STRT   T          ;START TIMER
```

Start Event Counter

Opcode	*Operand*
STRT	CNT

0 1 0 0	0 1 0 1

The test 1 (T1) pin is enabled as the event-counter input and the counter is started. The event-counter register is incremented with each high-to-low transition on the T1 pin.

Example: Initialize and start event counter. Assume overflow is desired with first T1 input.

```
STARTC: MOV    A,#0FFH    ;MOVE 'FF' HEX (ONES) TO
                          ;ACC
        MOV    T,A        ;MOVE ONES TO COUNTER
        EN     TCNTI      ;ENABLE COUNTER INTERRUPT
        STRT   CNT        ;ENABLE T1 AS COUNTER
                          ;INPUT AND START
```

Stop Timer/Event Counter

Opcode	*Operand*
STOP	TCNT

0 1 1 0	0 1 0 1

This instruction is used to stop both time accumulation and event counting.

Example: Disable interrupt, but jump to interrupt routine after eight overflows and stop timer. Count overflows in register 7.

```
START:   DIS    TCNTI     ;DISABLE TIMER INTERRUPT
         CLR    A         ;CLEAR ACC TO ZEROS
         MOV    T,A       ;MOVE ZEROS TO TIMER
         MOV    R7,A      ;MOVE ZEROS TO REG 7
MAIN:    STRT   T         ;START TIMER
         JTF    COUNT     ;JUMP TO ROUTINE 'COUNT'
                          ;IF TF=1 AND CLEAR TIMER FLAG
           .
           .
           .
         JMP    MAIN      ;CLOSE LOOP
COUNT:   INC    R7        ;INCREMENT REG 7
         MOV    A,R7      ;MOVE REG 7 CONTENTS TO ACC
         JB3    INT       ;JMUP TO ROUTINE 'INT' IF
                          ;ACC BIT 3 IS SET (REG 7=8)
         JMP    MAIN      ;OTHERWISE RETURN TO ROUTINE
                          ;'MAIN'
           .
           .
           .
INT:     STOP   TCNT      ;STOP TIMER
         JMP    7H        ;JUMP TO LOCATION 7
                          ;(TIMER INTERRUPT ROUTINE)
```

Clock Control

The test 0 (T0) pin can be used as a state clock output and tested directly by your program. See the MCS-48 user's manual for details.

Enable Clock Output

Opcode	*Operand*
ENT0	CLK

0	1	1	1	0	1	0	1

The test 0 pin is enabled to act as the clock output. This function is disabled by a system reset. The 8021 does not recognize this instruction.

Example:

```
ENTST0:  ENT0   CLK       ;ENABLE T0 AS CLOCK OUTPUT
```

Memory and Register Bank Controls

The following instructions allow you to control the interpretation of program memory references and references to data memory working registers. As noted in Chapter 1, memory and register bank selection is not possible on the 8021. It always refers to bank '0'.

Memory Bank Selection

The memory bank instructions let you specify your program memory address references to be in 'bank 0' (locations 0-2047) or 'bank 1' (locations 2048-4095). See Figure 1-1. These instructions toggle program counter bit 11, but not until the next branch from the main program (via a jump or call) begins execution.

	11	PROGRAM COUNTER	0		
Select Bank 0	0 0 1 0	1 0 0 0	1 0 1 0	=	Location 650
Select Bank 1	1 0 1 0	1 0 0 0	1 0 1 0	=	Location 2698

If a SEL MB instruction is issued before a CALL, it affects only the subroutine called. The return restores PC bit 11 to its previous value (see NOTE 1). A SEL MB issued before a jump instruction modifies PC bit 11 permanently.

NOTES

1. While PC bit 11 is restored on returning from a CALL, the 'designate bank' internal flip-flop (DBF) is not. This means you mustreset the DBF with another SEL before issuing another jump instruction.

2. When an interrupt service routine is executing, program counter bit 11 is held at zero. This means any service routine references must reside in memory bank 0. The select-memory-bank instructions should not be issued in an interrupt service routine.

The initial value of PC bit 11 is zero and memory bank 0 is selected.

Select Memory Bank 0

Opcode	*Operand*
SEL	MB0
1 1 1 0	0 1 0 1

PC bit 11 is set to zero. All references to program memory addresses fall within the range 0-2047. This instruction is not recognized by the 8021.

Example: Assume program counter contains 834H (2100D).

```
SEL  MBO        ;SELECT MEMORY BANK 0
JMP  $+20       ;JUMP TO LOCATION 48H (72D)
```

Select Memory Bank 1

Opcode	*Operand*
SEL	MB1

1	1	1	1	0	1	0	1

PC bit 11 is set to one. All references to program memory addresses fall within the range 2048-4095. This instruction is not recognized by the 8021.

Register Bank Selection

The register bank instructions let you specify whether references to registers 0-7 address data memory locations in 'bank 0' (locations 0-7) or 'bank 1' (locations 24-31). See Figure 1-2. These instructions toggle the register bank switch (PSW bit 4). The initial setting of this bit is zero.

Select Register Bank 0

Opcode	*Operand*
SEL	RBO

1	1	0	0	0	1	0	1

PSW bit 4 is set to zero. References to working registers 0-7 address data memory locations 0-7. This is the recommended setting for normal program execution. The 8021 does not recognize this instruction.

Select Register Bank 1

Opcode	*Operand*
SEL	RB1

1	1	0	1	0	1	0	1

PSW bit 4 is set to one. References to working registers 0-7 address data memory locations 24-31. This is the recommended setting for interrupt service routines, since locations 0-7 are left intact. The RETR instruction at the end of the interrupt service routine restores bit 4 of the PSW to the value it had at the time of the interrupt.

The 8021 does not recognize this instruction.

Example: Assume an external interrupt has occurred, control has passed to program memory location 3, and PSW bit 4 (register bank switch) was zero before the interrupt.

```
LOC3:   JMP     INIT      ;JUMP TO ROUTINE 'INIT' IF
                          ;INTERRUPT HAS OCCURRED
        .
        .
        .
INIT:   MOV     R7,A      ;MOVE ACC CONTENTS TO
                          ;LOCATION 7
        SEL     RB1       ;SELECT REG BANK 1
        MOV     R7,#0FAH ;MOVE 'FA' HEX TO LOCATION 31

        .
        .
        .
        SEL     RB0       ;SELECT REG BANK 0
        MOV     A,R7      ;RESTORE ACC FROM LOCATION 7
        RETR              ;RETURN – RESTORE PC AND PSW
                          ;4-7
```

The 'Null' Operation

The null operation uses one machine cycle, but no operation is performed. Its primary function is to reserve a program location for an instruction to be inserted later. It could also be used, like the timer, to synchronize your system.

The NOP Instruction

Opcode

NOP

0 0 0 0	0 0 0 0

No operation is performed. Execution continues with the following instruction.

TRANSFERRING PROGRAM CONTROL

Instructions in program memory are normally executed sequentially. Program control can be transferred out of the main line of code by an external or timer interrupt, or when a jump or call instruction is encountered.

An interrupt (if enabled) automatically transfers control to location 3 (for external interrupts) or location 7 (for timer overflows), and is essentially a call to an interrupt service subroutine. The program counter and PSW bits 4-7 are saved in the stack.

Your program can also contain other subroutines to perform frequently-executed code. Control is passed to these subroutines by the CALL instruction, which also saves the program counter and PSW bits 4-7.

Control is returned from an interrupt service routine or other subroutine to the main program by the RET and RETR instructions. RET restores only the program counter; RETR restores both the program counter and PSW bits 4-7.

The jump instructions alter the contents of the program counter without saving PC or PSW information. Jumps can be specified subject to certain conditions (such as the setting of a flag), or can be made unconditional.

All conditional jumps and the JMPP instruction limit the range of a jump to the current 256-location page (that is, alter PC bits 0-7). The JMP and CALL instructions allow program control to be transferred within a 2K memory bank (that is, alter PC bits 0-10). This range can be extended to 4K by toggling PC bit 11 with the SEL MB instructions. A SEL MB preceding a CALL instruction is valid only for the duration of the subroutine; a SEL MB preceding a jump remains in effect until changed by your program.

Jump instructions with 8-bit operands imply a destination address expressable in 12 bits. All 8-bit addresses are valid; 12-bit destination addresses are valid if the jump instruction and destination reside on the same page. If a conditional jump or JMPP begins in location 255 of a page, it must reference a destination on the following page. Any jump instruction beginning in location 2047 or 4095 is invalid. A CALL cannot begin in locations 2046-2047 or 4094-4095.

All control transfer and return instructions require two cycles for execution.

Subroutine Call/Return Operations

Subroutines are entered and exited using the CALL, RET, and RETR instructions.

Subroutine Call

Opcode	*Operand*
CALL	address

10 8		7 0
addr	1 0 1 0 0	addr

The program counter and PSW bits 4-7 are saved in the stack. The stack pointer (PSW bits 0-2) is updated. Program control is then passed to the location specified by 'address.' PC bit 11 is determined by the most recent SEL MB instruction. PC bits 10-11 must always be '0' for the 8021 or a 'range' error (R) results.

Execution continues at the instruction following the CALL upon return from the subroutine.

Example: Add three groups of two numbers. Put subtotals in locations 50, 51 and total in location 52.

```
          MOV   R0,#50   ;MOVE '50' DEC TO ADDRESS
                         ;REG 0
BEGADD:   MOV   A,R1     ;MOVE CONTENTS OF REG 1 TO
                         ;ACC
          ADD   A,R2     ;ADD REG 2 TO ACC
          CALL  SUBTOT   ;CALL SUBROUTINE 'SUBTOT'
          ADD   A,R3     ;ADD REG 3 TO ACC
          ADD   A,R4     ;ADD REG 4 TO ACC
          CALL  SUBTOT   ;CALL SUBROUTINE 'SUBTOT'
          ADD   A,R5     ;ADD REG 5 TO ACC
          ADD   A,R6     ;ADD REG 6 TO ACC
          CALL  SUBTOT   ;CALL SUBROUTINE 'SUBTOT'
           .
           .
           .
SUBTOT:   MOV   @R0,A    ;MOVE CONTENTS OF ACC TO
                         ;LOCATION ADDRESSED BY  REG 0

          INC   R0       ;INCREMENT REG 0
          RET            ;RETURN TO MAIN PROGRAM
```

Return Without PSW Restore

Opcode

RET

1	0	0	0	0	0	1	1

The stack pointer (PSW bits 0-2) is decremented. The program counter is then restored from the stack. PSW bits 4-7 are not restored.

Return With PSW Restore

Opcode

RETR

1	0	0	1	0	0	1	1

The stack pointer is decremented. The program counter and bits 4-7 of the PSW are then restored from the stack. Note that RETR should be used to return from an interrupt, but should not be used *within* the interrupt service routine as it would clear the interrupt in process.

This instruction is not recognized by the 8021.

Jump Instructions

The MCS-48 instruction set includes two unconditional jumps and 13 conditional jumps (in addition to the DJNZ instruction described earlier in this chapter). Only one jump instruction, JMP, alters PC bits 0-10. The others affect only PC bits 0-7 and, therefore, must address a location within the current 256-location page.

Unconditional Jumps

The JMP unconditional jump allows you to cross page boundaries; JMPP is limited to the current page. JMP addresses program memory locations directly; JMPP addresses program memory locations indirectly through the accumulator. Indirection is indicated by prefixing the accumulator reference with a 'commercial at' (@).

Direct Jump Within 2K Block

Opcode	*Operand*
JMP	address

10 8		7 0
addr	0 0 1 0 0	addr

Bits 0-10 of the program counter are replaced with the directly-specified address. The setting of PC bit 11 is determined by the most recent SELECT MB instruction. PC bits 10-11 must always be '0' for the 8021 or a 'range' error (R) results.

Examples:

```
JMP   SUBTOT            ;JUMP TO SUBROUTINE 'SUBTOT'
JMP   $-6               ;JUMP TO INSTRUCTION SIX LOCATIONS
                        ;BEFORE CURRENT LOCATION
JMP   2FH               ;JUMP TO ADDRESS '2F' HEX
```

Indirect Jump Within Page

Opcode	*Operand*
JMPP	@A

1 0 1 1	0 0 1 1

The contents of the program memory location pointed to by the accumulator are substituted for the 'page' portion of the program counter (PC bits 0-7).

Example: Assume accumulator contains 0FH.

```
JMPPAG:  JMPP @A        ;JUMP TO ADDRESS STORED IN
                        ;LOCATION 15 IN CURRENT PAGE
```

Conditional Jumps

The following jumps are executed only if a specific condition is satisfied. All jumps occur within the current page.

Jump If Carry Is Set

Opcode	*Operand*
JC	address

1 1 1 1	0 1 1 0	address

Control passes to the specified address if the carry bit is set to one.

Example:

```
JC1:    JC   OVFLOW          ;JUMP TO 'OVFLOW' ROUTINE IF C=1
```

Jump If Carry Is Not Set

Opcode	*Operand*
JNC	address

1 1 1 0	0 1 1 0	address

Control passes to the specified address if the carry bit is not set, that is, equals zero.

Example:

```
JC0:    JNC   NOVFLO         ;JUMP TO 'NOVFLO' ROUTINE IF C=0
```

Jump If Accumulator Is Zero

Opcode	*Operand*
JZ	address

1 1 0 0	0 1 1 0	address

Control passes to the specified address if the accumulator contains all zeros when this instruction is executed. Accumulator contents are monitored continuously.

Example:

```
JACC0:     JZ   0A3H        ;JUMP TO LOCATION 'A3' HEX
                            ;IF ACC VALUE IS ZERO
```

Jump If Accumulator Is Not Zero

Opcode	*Operand*
JNZ	address

1 0 0 1	0 1 1 0	address

Control passes to the specified address if the accumulator contents are nonzero when this instruction is executed. Accumulator contents are monitored continuously.

Example:

```
JACCN0:     JNZ  0ABH        ;JUMP TO LOCATION 'AB' HEX
                             ;IF ACC VALUE IS NONZERO
```

Jump If Flag 0 Is Set

Opcode	*Operand*
JF0	address

1 0 1 1	0 1 1 0	address

Control passes to the specified address if flag 0 is set to one. This instruction is not recognized by the 8021.

Example:

```
JF0IS1:   JF0   TOTAL        ;JUMP TO 'TOTAL' ROUTINE IF
                             ;F0=1
```

Jump If Flag 1 Is Set

Opcode	*Operand*
JF1	address

0 1 1 1	0 1 1 0	address

Control passes to the specified address if flag 1 is set to one. This instruction is not recognized by the 8021.

Example:

```
JF1IS1:   JF1   FILBUF       ;JUMP TO 'FILBUF' ROUTINE
                             ;IF F1=1
```

Jump If Test 0 Is High

Opcode	*Operand*
JT0	address

0 0 1 1	0 1 1 0	address

Control passes to the specified address if the test 0 signal is high (=1). This instruction is not recognized by the 8021.

Example:

```
JT0HI:     JT0  53          ;JUMP TO LOCATION 53 DEC IF
                            ;T0=1
```

Jump If Test 0 Is Low

Opcode	*Operand*
JNT0	address

0 0 1 0	0 1 1 0	address

Control passes to the specified address if the test 0 signal is low (=0). This instruction is not recognized by the 8021.

Example:

```
JT0LOW:     JNT0  60        ;JUMP TO LOCATION 60
                            ;DEC IF T0=0
```

Jump If Test 1 Is High

Opcode	*Operand*
JT1	address

0 1 0 1	0 1 1 0	address

Control passes to the specified address if the test 1 signal is high (=1).

Example:

```
JT1HI:     JT1  COUNT       ;JUMP TO 'COUNT' ROUTINE
                            ;IF T1=1
```

Jump If Test 1 Is Low

Opcode	*Operand*
JNT1	address

0	1	0	0	0	1	1	0	address

Control passes to the specified address if test 1 signal is low (=0).

Example:

```
JT1LOW:  JNT1    NOCNT        ;JUMP TO 'NOCNT' ROUTINE
                              ;IF T1=0
```

Jump If Timer Flag Is Set

Opcode	*Operand*
JTF	address

0	0	0	1	0	1	1	0	address

Control passes to the specified address if the timer flag is set to one, that is, the timer/counter register has overflowed. Testing the timer flag resets it to zero. (This overflow initiates an interrupt service sequence if the timer-overflow interrupt is enabled.)

Example:

```
JTF1:    JTF     TIMER        ;JUMP TO 'TIMER' ROUTINE
                              ;IF TF=1
```

Jump If Interrupt Input Is Low

Opcode	*Operand*
JNI	address

1	0	0	0	0	1	1	0	address

Control passes to the specified address if the interrupt input signal is low (=0), that is, an external interrupt has been signalled. (This signal initiates an interrupt service sequence if the external interrupt is enabled.) The 8021 does not recognize this instruction.

Example: The JNI instruction is used to control a test input.

```
         DIS     I            ;DISABLE EXTERNAL INTERRUPT
         JNI     TRUE         ;JUMP TO 'TRUE' ROUTINE
                              ;IF I=0 (TEST IS TRUE)
         JMP     $-2          ;LOOP TO JNI TEST
```

Jump If Accumulator Bit Is Set

Opcode	*Operand*	
JBb	address	b=0-7

b b b 1	0 0 1 0	address

Control passes to the specified address if accumulator bit 'b' is set to one. The 8021 does not recognize this instruction.

Example:

```
JB4IS1:    JB4   NEXT          ;JUMP TO 'NEXT' ROUTINE
                               ;IF ACC BIT 4=1
```

SAMPLE PROGRAMS

The following examples demonstrate addition, subtraction, multiplication, and number comparison using 8-bit, 16-bit, and BCD quantities. Analog/digital conversion and a keyboard scan are also demonstrated.

Addition With 8-Bit Quantities

Add 8-bit symbolic values ADDEND and AUGEND and place their sum in Register 7.

```
ADD8:      MOV    A,#ADDEND
           ADD    A,#AUGEND
           MOV    R7,A
```

Addition With 16-Bit Quantities

Add two 16-bit numbers and store their sum in registers 6 (high-order byte) and 7 (low-order byte).

```
ADD16:     MOV    A,#ADDLOW
           ADD    A,#AUGLOW
           MOV    R7,A
           MOV    A,#ADDHI
           ADDC   A,#AUGHI    ;INCLUDE OVERFLOW FROM
                              ;PREVIOUS ADD IN ADDITION
           MOV    R6,A
```

Addition With BCD Quantities

Add the BCD number whose LSD is at location BETA to the BCD number whose LSD is at location ALPHA and store the result in ALPHA. Length of number is 'COUNT' digit pairs. For this example, assume both numbers are the same length and have an even number of digits (or the most-significant digit is zero, if *odd*).

```
ADDBCD:    MOV    R0,#ALPHA       ;AUGEND, SUM LSD
                                  ;LOCATION IN REG 0
           MOV    R1,#BETA        ;ADDEND LOCATION
                                  ;IN REG 1
           MOV    R2,#COUNT       ;LOOP COUNTER IN
                                  ;REG 2
           CLR    C
LOOP:      MOV    A,@R0           ;ADD ROUTINE
           ADDC   A,@R1
           DA     A
           MOV    @R0,A           ;STORE SUM
           DEC    R0              ;DECREMENT ADDRESS
                                  ;REGS
           DEC    R1
           DJNZ   R2,LOOP         ;LOOP CONTROL
```

Subtraction With 8-Bit Quantities

Subtract 8-bit subtrahend from 8-bit minuend using two's complement addition and store difference in register 7.

```
SUB8:      MOV    A,#SUBHND
           CPL    A               ;ONE'S COMPLEMENT A
           INC    A               ;TWO'S COMPLEMENT A
           ADD    A,#MINEND
           MOV    R7,A
```

Subtraction With 16-Bit Quantities

Subtract two 16-bit numbers and store their difference in registers 3 (high-order byte) and 4 (low-order byte). Note the use of ADD, rather than INC, to form the two's complement numbers; INC does not affect the carry bit.

```
SUB16:     MOV    A,#SUBLOW
           CPL    A
           ADD    A,#1            ;FORM TWO'S COMPLEMENT
           MOV    R4,A            ;STORE TEMP SUBLOW COMP
           MOV    A,#SUBHI
           CPL    A
           ADDC   A,#0            ;PICK UP OVERFLOW AND
                                  ;FORM TWO' S COMPLEMENT
           MOV    R3,A            ;STORE TEMP SUBHI COMP
           MOV    A,R4            ;BEGIN ADDITION
           ADD    A,#MINLOW
           MOV    R4,A            ;STORE LOW-ORDER DIFF
           MOV    A,R3
           ADDC   A,#MINHI
           MOV    R3,A            ;STORE HIGH-ORDER DIFF
```

Multiplication (8 X 8 Bits, 16-Bit Product)

Multiply two 8-bit numbers and store the 16-bit product in registers 2 and 3. Note than nine shifts through the accumulator are required to justify the product correctly.

```
MPY8X8: MOV   R5,#9          ;8 + 1 IN LOOP COUNTER
        MOV   R6,#MCAND      ;MULTIPLICAND IN REG 6
        MOV   R3,#MPLIER     ;MULTIPLIER, LOW PARTIAL
                             ;PRODUCT IN REG 3
        CLR   A
        CLR   C
LOOP:   RRC   A              ;ROTATE
        XCH   A,R3           ;  CARRY, ACC, REG 3
        RRC   A              ;     RIGHT
        XCH   A,R3           ;        ONE BIT
        JNC   NOADD          ;TEST CARRY
        ADD   A,R6
NOADD:  DJNZ  R5,LOOP        ;9 SHIFTS TO JUSTIFY
        MOV   R2,A           ;STORE HIGH PARTIAL
                             ;PRODUCT
```

Compare Memory to Accumulator

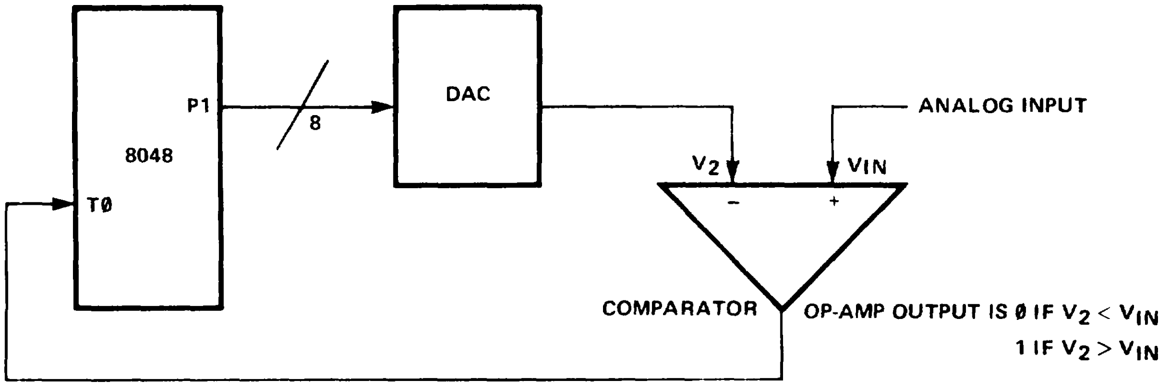
Make an unsigned comparison between the contents of a memory location and the accumulator. Save original accumulator contents temporarily in register 5.

```
COMPAR: MOV   R5,A
        MOV   R0,#MEM        ;ADDRESS OF NUMBER TO BE
                             ;COMPARED
        CPL   A
        INC   A
        ADD   A,@R0          ;ACC CONTENTS DESTROYED
        JZ    EQUAL          ;ACC = MEM
        JNC   ACCGT          ;ACC GREATER THAN MEM
        JC    ACCLT          ;ACC LESS THAN MEM
```

Analog/Digital Conversion

Construct an A/D converter from a D/A converter, a comparator op-amp, and a successive-approximation software routine.

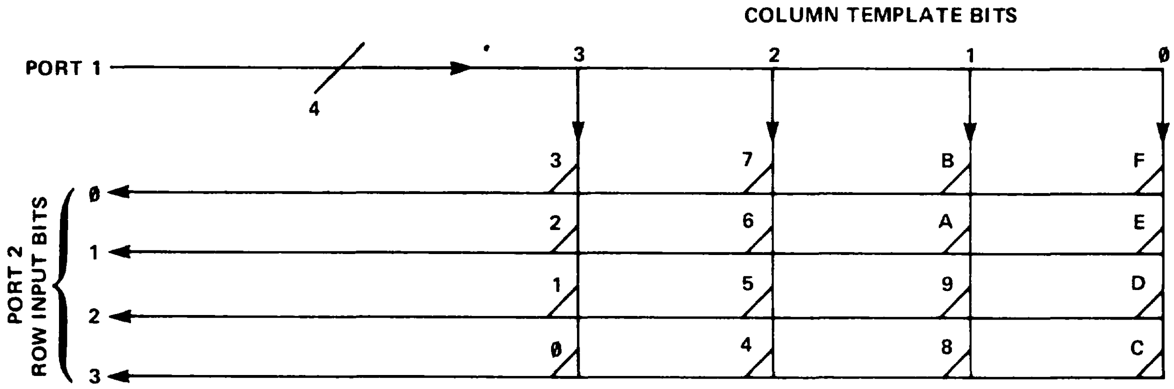
The 8048 sends eight bits of data to the D/A converter via output port 1. The output of the D/A converter is compared to the 'analog input' being converted. The result of the comparison (0 if DAC output is lower, 1 if higher) is sent back to the 8048 via the T0 input pin for handling. This procedure lets the 8048 estimate the proper digital representation of the analog input by testing the most significant bit, keeping it if the input says 'still too low' or dropping it if the input says 'too high now.' From this point, each bit is tested in order of significance and either kept or discarded.



```
ADCON:  MOV     R7,#8H          ;COUNTER R7=8
        CLR     A               ;CLEAR A,R5,R6
        MOV     R5,A
        MOV     R6,A
        CLR     C
        CPL     C               ;SET CARRY
LOOP:   MOV     A,R5            ;ROTATE TEST BIT
        RRC     A               ;   RIGHT FROM MSB
        MOV     R5,A            ;       TO LSB
        ORL     A,R6            ;ADD TO R6 VALUE
        OUTL    P1,A            ;TEST NEW VALUE
        JT0     DROP            ;IF T0=1, DROP NEW VALUE
        MOV     R6,A            ;IF T0=0, SAVE NEW VALUE
DROP:   DJNZ    R7,LOOP         ;TEST NEXT BIT
```

Matrix Keyboard Scan (4X4)

A keyboard is arranged such that any key pressed in any of four vertical columns returns a recognizable signal to the microprocessor. When the key is pressed, its signal goes low and a 0 is returned to the processor. (For example, pressing key 9 returns the bit pattern 1011 when the template for vertical column 1 is operative.)



The microprocessor scans the keyboard until it detects a low signal (pressed key). This triggers a check loop to ensure no other key has been pressed. When this check has been completed, the processor stores the value of the key. If two keys are pressed, the routine ignores both and starts again at its entry point.

Register and port assignments are as follows:

R0	Key counter
R1	Address for key storage
R2	Column template
R3	Row counter
R4	Intermediate key storage
P1	Column template output
P2	Keyboard input to processor

```
SCAN:   CALL    SCANKY        ;SCAN MATRIX FOR KEY
        XCH     A,R0          ;SAVE KEY ID
        MOV     R4,A
        XCH     A,R0
        CALL    CHECK         ;CHECK FOR 2ND KEY
        MOV     A,R4          ;COMPARE NEW/OLD KEYS
        XRL     A,R0
        JNZ     SCAN          ;RESTART IF 2 KEYS PRESSED
        MOV     R1,#STORKY    ;ADDR FOR STORING KEY
        MOV     A,R0
        MOV     @R1,A         ;STORE HOME KEY
```

```
WAIT:    MOV   A,R2        ;WAIT FOR KEY
         OUTL  P1,A        ;  TO BE
         IN    A,P2        ;      RELEASED
         CPL   A
         JNZ   WAIT
         JMP   DONE        ;DONE WHEN KEY RELEASED
SCANKY:  MOV   R0,#0FH     ;KEY COUNTER
         MOV   R2,#7FH     ;COLUMN TEMPLATE
NEWCOL:  MOV   R3,#4H      ;ROW COUNTER
         MOV   A,R2        ;PUT TEMPLATE INTO ACC
         RL    A           ;ROTATE TEST BIT INTO POSITION
         OUTL  P1,A        ;OUTPUT ROTATED TEMPLATE
         MOV   R2,A        ;SAVE ROTATED TEMPLATE
         IN    A,P2        ;INPUT (KEY PRESSED)
COLUMN:  RRC   A           ;CHECK INPUT
         JC    CHECK       ;BRANCH IF ROW=1 (HIGH)
         RET               ;RETURN TO MAIN IF ROW=0 (LOW)
CHECK:   DEC   R0          ;DECREMENT KEY COUNT
         DJNZ  R3, COLUMN  ;DECREMENT COLUMN COUNT
         MOV   A,R0        ;IF ACC IS NOT ZERO,
         JNZ   NEWCOL      ;  SCAN NEXT COLUMN
         JMP   SCANKY      ;START NEW SCAN
```

4. UPI-41 ASSEMBLY LANGUAGE INSTRUCTIONS

In Chapter 1 we described the functional and hardware differences between the 8048 and 8041 microcomputers. This chapter lists the instruction set differences.

Most of the instructions described in Chapter 3 apply to the UPI-41 microcomputers (8041/8741) also. However, ten MCS-48 instructions are deleted from the UPI-41 instruction set (treated as undefined), two are interpreted differently on the UPI-41, and the UPI-41 instruction set includes four additional instructions for performing the handshaking protocol.

The 8048/41 assemblers normally assume you are using the MCS-48 instruction set. If you wish to use the UPI-41 instruction set, you must first issue the assembler control 'MOD41.' See Part Two for details.

DELETED 8048 INSTRUCTIONS

As was mentioned in Chapter 1, the 8041's BUS port is required for the master-slave handshaking protocol. Instructions requiring or defining the use of this port, namely the instructions used to access or define external data or program memory, plus I/O instructions addressing the BUS, are not recognized by the 8041. These instructions are:

```
MOVX  @Rr,A       (Access external data memory)
MOVX  A,@Rr

SEL   MB0         (Define external program memory)
SEL   MB1

INS   A,BUS       (I/O operations using BUS)
ANL   BUS,#data
ORL   BUS,#data
OUTL  BUS,A
```

The external interrupt function is also committed to the master processor interface. Therefore the following instruction is also unrecognized:

```
JNI  addr         (Jump if external interrupt pin is low)
```

The T0 pin can function only as a test input. The following instruction is unrecognized:

```
ENT0  CLK
```

Finally, a CALL or JMP to pages 4-7 (that is, beyond address 1023) causes a range error.

REINTERPRETED INSTRUCTIONS

When the master processor fills the 8041's data bus buffer (DBB) with data, it can cause an interrupt (as a check against more data being transferred before the buffer is cleared). Like the external interrupt on the 8048, this interrupt forces a call to location 3. The data transfer interrupt is also enabled and disabled by the same instructions used to enable and disable external interrupts on the 8048:

```
EN   I
DIS  I
```

ADDED INSTRUCTIONS

The UPI-41 instruction set includes two instructions for transferring data to/from the DBB and the 8041's accumulator. It also includes two instructions for testing the input buffer (IBF) and output buffer (OBF) flags in the 8041 status register.

Data Transfer Instructions

Input DBB Contents to Accumulator

Opcode	*Operands*
IN	A,DBB

0 0 1 0	0 0 1 0

This instruction loads the 8041's accumulator with the contents of the data bus buffer. It also clears the input buffer flag (which was set when the master computer filled the DBB with input data). This flag is initially cleared.

NOTE

This instruction cannot be used to read back data previously output to the DBB. Correct operation of 'IN A,DBB' is guaranteed only if IBF=1 and OBF=0.

Output Accumulator Contents to DBB

Opcode	*Operands*
OUT	DBB,A

0 0 0 0	0 0 1 0

Note that the encoding of this instruction is the same as for the OUTL BUS,A instruction. The contents of the accumulator are stored in the data bus buffer and the output buffer flag is set. This flag is initially cleared.

Flag Test Instructions

These two instructions transfer program control conditionally depending on the setting of IBF and OBF. IBF is set when the data bus buffer is filled by the master processor; OBF is set when the DBB is filled with data to be transferred to the master processor.

Note that program control can only be transferred within the current 256-location page.

Jump If IBF Is Not Set

Opcode	*Operand*
JNIBF	address

1 1 0 1	0 1 1 0	address

This is a 2-cycle instruction. Control passes to the specified address if IBF is zero, that is, if the DBB is not filled with input data.

Example:

```
LODBUF:  JNIBF    INPUT       ;JUMP TO 'INPUT' ROUTINE
                              ;IF IBF=0
```

Jump If OBF Is Set

Opcode	*Operand*
JOBF	address

1 0 0 0	0 1 1 0	address

This is a 2-cycle instruction. Note that the encoding of this instruction is the same as for the JNI conditional jump. Control passes to the specified address if OBF is set to one, that is, if the DBB is filled with output data.

Example:.

```
         JOBF     OUTPUT      ;JUMP TO 'OUTPUT' ROUTINE
                              ;IF OBF=1
```

5. ASSEMBLER DIRECTIVES

This chapter and Chapter 6 describe the assembler directives used to control the MCS-48/UPI-41 assemblers in their generation of object code. These directives are written in the same format as MCS-48 instructions, in general, and can be interspersed throughout your assembly language program.

Unlike assembly language instructions, however, they produce no executable object code.

5

Assembler directives can be divided functionally as follows:

- Location counter control
  - ORG
- Symbol definition
  - EQU
  - SET
- Data definition
  - DB
  - DW
- Memory reservation
  - DS
- Conditional assembly
  - IF
  - ELSE
  - ENDIF
- Macro operations
  - MACRO
  - LOCAL
  - ENDM
  - Macro call
  - REPT
  - IRP
  - IRPC
  - EXITM
- Assembler termination
  - END
- End-of-tape indication
  - EOT

Macro operations are discussed separately in the next chapter.

One notable format difference between assembler directives and MCS-48 instructions involves the 'label' field. This field is always optional and is always terminated by a colon (:) in MCS-48 instructions. The same is generally true of assembler directives, but three directives (EQU, SET, MACRO) *require* the name of the symbol or macro being defined to be present in the label field, and this name *cannot* be terminated by a colon. The LOCAL and ENDM assembler directives *prohibit* inclusion of the label field.

LOCATION COUNTER CONTROL

The location counter performs the same function for the assembler as the program counter performs during program execution. It tells the assembler the next memory location available for instruction or data assembly.

The location counter can be set by the ORG (origin) directive. See also the discussion of the END directive in the section 'Assembler Termination,' later in this chapter.

ORG Directive

Label	*Opcode*	*Operand*
optional:	ORG	expression

The location counter is set to the value of 'expression,' which must evaluate to a valid 12-bit program memory address. If 'expression' is a symbol, the symbol must be previously defined. The next machine instruction or data item is assembled at the address specified. If no ORG is included before the first instruction or data byte in your program, assembly begins at location zero.

Your program can include any number of ORG statements. Multiple ORGs need not be listed in ascending order, but failure to do so creates potential memory overlap problems.

Example:

```
PAG1:  ORG   0FFH      ;ORG ASSEMBLER TO LOCATION
                       ;'FF' HEX (255 DEC)
```

SYMBOL DEFINITION

Symbol names appearing as labels in MCS-48 instructions are assigned values automatically during the assembly process. The value in this case is the value in the location counter when the labeled instruction is assembled.

Other symbols are defined and assigned arbitrary values using the EQU and SET directives. Symbols defined using EQU cannot be redefined during assembly; those defined by SET can be assigned new values by subsequent SET directives.

The symbol name required in the label field of an EQU or SET directive is *not* terminated by a colon.

Symbols defined by EQU and SET have global scope, that is, they can be referenced from any instruction in your program. If a symbol appears only in the body of a macro definition, however, it should be given limited scope using the LOCAL directive. (See Chapter 6.)

EQU Directive

Label	*Opcode*	*Operand*
name	EQU	expression

The symbol 'name' is created and assigned the value of 'expression.' This 'name' cannot appear in the label field of another EQU directive, that is, it is not redefinable.

Example:

```
ONES    EQU    0FFH          ;CREATE SYMBOL 'ONES' WITH
                             ;BINARY VALUE 11111111
```

SET Directive

Label	*Opcode*	*Operand*
name	SET	expression

The symbol 'name' is assigned the value of 'expression.' Wherever the symbol name is encountered in the assembly, this value is used unless 'name' is assigned a new value by another SET directive.

DATA DEFINITION

The DB (define byte) and DW (define word) directives enable you to specify data in your program. Data can be specified in the form of 8-bit or 16-bit values, or as a text string.

DB Directive

Label	*Opcode*	*Operands*
optional:	DB	expression(s) or string(s)

The operand field of the DB directive can contain a list of expressions and/or text strings with the list items separated by commas. The list can contain up to eight total elements, but elements containing expressions can reduce this maximum allowance.

Expressions must evaluate to 1-byte (8-bit) numbers. This provides a range of –256 to +255 (all ones or all zeroes in the high-order byte of the internal representation). Strings can extend over an arbitrary number of bytes. The bytes assembled for the DB directive are stored consecutively in available memory starting at the address in the location counter.

Example:

```
DATA:    DB   ‘STRING 1’, ‘STRING 2’, 3, 4
QUOTE:   DB   ‘THIS IS A QUOTE’’’
```

DW' Directive

Label	Opcode	Operands
optional:	**DW**	expression(s) or string constant(s)

The operand field of the DW directive can contain a list of expressions and/or 1-byte or 2-byte string constants. List items are separated by commas. The list can contain up to eight total elements, but elements containing expressions can reduce this maximum allowance.

Expressions must evaluate to 1-word (16-bit) numbers. The high-order eight bits of the 16-bit value are assembled at the address in the location counter; the low order eight bits are assembled at the next higher location.

Strings are limited to one or two characters. In the case of a single-character string, the high-order eight bits are filled with zeros.

Examples:

```
ADDR:    DW   FIRST, LAST
PAGES:   DW   0,0100H,0200H,0300H
STRS:    DW   ‘AB’, ‘CD’
```

MEMORY RESERVATION

A block of program memory can be reserved using the DS (define storage) directive. No data is assembled into these locations and no assumptions can be made about their initial contents when your program is loaded.

DS Directive

Label	Opcode	Operand
optional:	**DS**	expression

‘Expression’ specifies the number of locations to be reserved for data storage. This block of memory locations is reserved by incrementing the location counter by the value of ‘expression’. This value must be absolute. Any symbol appearing in the operand field must be previously defined.

If the optional label is present, it is assigned the starting value of the location counter (before incrementing), and thus references the starting address of the reserved block.

If the value of 'expression' is zero, no memory is reserved, but the label is assigned the current value of the location counter.

Example:

```
        TTYBUF:  DS  72            ;RESERVE 72 LOCATIONS AS A
                                   ;TERMINAL OUTPUT BUFFER
```

CONDITIONAL ASSEMBLY

The IF, ELSE, and ENDIF directives enable you to assemble portions of your program conditionally, that is, only if certain conditions that you specify are satisfied.

These directives are used in coordination, and consequently are not separated in the following description.

IF, ELSE, and ENDIF Directives

Label	*Opcode*	*Operand*
optional:	IF	expression
optional:	ELSE	---
optional:	ENDIF	---

The assembler evaluates the 'expression' in the operand field of the IF directive. If bit 0 of the resulting value is one (TRUE), all instructions between the IF directive and the next ELSE or ENDIF directive are assembled. If bit 0 is zero (FALSE), these instructions are ignored. (A TRUE expression evaluates to 0FFFFH and FALSE to 0H, and consequently only one bit need be tested.)

ELSE is the converse of IF. If bit 0 of 'expression' is zero, all instructions between ELSE and the next ENDIF are assembled. If bit 0 is one, these instructions are ignored.

All statements included between an IF directive and its associated ENDIF are defined as an IF-ENDIF block. These blocks can be nested to eight levels. The ELSE directive is optional and only one ELSE can be included in an IF-ENDIF block.

NOTE

Data appearing in the operand field of an ELSE or ENDIF directive causes an error. Any symbol used in 'expression' must be previously defined. Conditional blocks cannot be split across macro definitions as nesting errors would result (that is, a conditional assembly block initiated in a macro definition must have its matching ENDIF in the same macro definition).

Examples:

```
          COND1:          IF TYPE EQ 0
                          .
                          .                      ;ASSEMBLED IF 'TYPE = 0'
                          .                      ;IS TRUE
                          .
                          ENDIF

          COND2:          IF TYPE EQ 0
                          .
                          .                      ;ASSEMBLED IF 'TYPE = 0'
                          .                      ;IS TRUE
                          .
                          ELSE
                          .
                          .
                          .                      ;ASSEMBLED IF 'TYPE = 0'
                          .                      ;IS FALSE
                          .
                          ENDIF


          COND3:          IF TYPE EQ 0
                          .
                          .                      ;ASSEMBLED IF 'TYPE = 0'
                          .                      ;IS TRUE
                          .
                          IF MODEL EQ 1
                          .
              LEVEL       .                      ;ASSEMBLED IF 'TYPE = 0'
                1         .                      ;AND 'MODEL = 1' ARE BOTH
                          .                      ;TRUE
                          ENDIF
                          ELSE
                          .
 LEVEL                    .                      ;ASSEMBLED IF 'TYPE = 0'
   2                      .                      ;IS FALSE
                          .
                          IF MODEL EQ 2
                          .
                          .                      ;ASSEMBLED IF 'TYPE = 0'
                          .                      ;IS FALSE AND 'MODEL = 2'
                          .                      ;IS TRUE
              LEVEL       ELSE
                1         .
                          .                      ;ASSEMBLED IF 'TYPE = 0'
                          .                      ;AND 'MODEL = 2' ARE BOTH
                          .                      ;FALSE
                          ENDIF
                          ENDIF
```

ASSEMBLER TERMINATION

The END directive terminates assembler execution. Its interpretation can differ slightly, depending on whether you are using the diskette-resident or paper-tape resident version of the assembler.

END Directive

Label	*Opcode*	*Operand*
optional:	END	expression

The END directive identifies the end of the source program and terminates each pass of the assembler. Only one END directive can appear in your program and it must be the last source line of the program. When source files are combined using the INCLUDE control (Chapter 8), there are no restrictions on which source file contains the END.

If 'expression' is specified in the operand field, its value is used as the program execution starting address. If no 'expression' is given, the starting address is zero.

Example:

```
        END   START           ;EXECUTION BEGINS AT THE
                              ;ADDRESS LABELED 'START'
```

When the paper-tape resident assembler is used, the END directive terminates each assembler pass, then causes the assembler to prompt you for the next pass to be executed.

END-OF-TAPE INDICATION

The EOT directive allows you to specify the physical end of paper tape to simplify assembly of multiple-tape source programs.

EOT Directive

Label	*Opcode*	*Operand*
optional:	EOT	---

When EOT is recognized by the assembler, the message 'NEXT TAPE' is sent to the console and the assembler pauses. After the next tape is loaded, a 'space bar' character received at the console signals continuation of the assembly.

Data in the operand field causes an error.

6. MACROS

The paper-tape-resident and ROM-resident 8048/41 assemblers do not support macro operations. If you are using either of these versions of the assembler, you can ignore this chapter.

Macro directives are extremely powerful tools. Properly used, they can increase program readability and efficiency. We strongly suggest that you become familiar with these directives and use them to tailor programs to suit your specific needs.

INTRODUCTION TO MACROS

Why Use Macros?

A macro is essentially a facility for replacing one set of parameters with another. In developing your program, you will frequently find that many instruction sequences are repeated several times, with only certain parameters changed. Rather than rewrite this code each place it occurs, you might prefer to code the sequence once (inserting dummy parameters in the fields subject to change) and later call this code with a single instruction wherever it is needed (replacing the dummy parameters with actual values at that time). The macro facility of the 8048/41 assemblers provides this capability and offers several advantages over writing code repetitiously.

- The tedium of frequent rewrite (and increased probability of error) is reduced.
- Symbols used in macros can be given limited scope, reducing the possibility of duplicate symbols.
- An error detected in a macro need be corrected in only one code segment, reducing debugging time.
- Duplication of effort between programmers can be reduced. Useful functions can be collected in a system macro library.

In addition, macros can be used to improve program readability and especially to create structured programs. Using macros to segment code blocks provides clear program notation and simplifies tracing the logic flow of the program.

What is A Macro?

A macro can be described as 'a routine *defined* in a formal sequence of prototype instructions that, when *called* within a program, results in the replacement of each such call with a code *expansion* consisting of the sequence of actual instructions represented.

The concepts of macro definition, call, and expansion can be illustrated by a typical business form letter, where the 'prototype instructions' consist of preset text (not to be confused with an *actual* MCS-48 macro). For example, we could define a macro CNFIRM with the text

> 'Air Flight welcomes you as a passenger.
> Your flight number FNO leaves at DTIME and arrives in DEST at ATIME.'

This macro has four dummy parameters to be replaced, when the macro is called, by the actual flight number, departure time, destination, and arrival time. Thus the macro call might look like

```
CNFIRM     123,      '10:45',      'Ontario',     '11:52'
```

A second macro, CAR, could be called if the passenger has requested that a rental car be reserved at the destination airport. This macro might have the text

> 'Your automobile reservation has been confirmed with MAKE rental car agency.'

Finally, a macro GREET could be defined to specify the passenger name.

> 'Dear NAME:'

The entire text of the business letter (source file) would then look like

```
GREET 'Ms. Scannell'
CNFIRM 123, '10:45', 'Ontario', '11:52'
CAR 'Blotz'
We trust you will enjoy your flight.

Sincerely,
```

When this source file is passed through a macro processor, the macro calls are expanded to produce the following letter.

> *Dear Ms. Scannell:*
>
> *Air Flight welcomes you as a passenger. Your flight number 123 leaves at 10:45 and arrives in Ontario at 11:52. Your automobile reservation has been confirmed with Blotz rental car agency.*
>
> *We trust you will enjoy your flight.*
>
> *Sincerely,*

Macros Vs. Subroutines

At this point you may be wondering how macros differ from subroutines called by the 8048 CALL instruction. Both aid program structuring and reduce coding of frequently-executed routines.

One distinction between the two is that macros generate in-line code while subroutines necessarily branch to another part of your program. Also, macro parameters are evaluated at assembly time; the variables used in subroutines are evaluated only during program execution (that is, at run time). Macros furthermore, can operate with data as well as program instructions.

Determining which of these facilities to use in a given programming situation is not always an obvious decision. For example, a choice to reduce the overall program size using subroutines may cause the program to run more slowly. Very long routines may best be handled as subroutines, while routines including many parameters are best coded as macros. Or you may find a combination of the two (such as a macro that calls a subroutine) to be your best solution.

Your decision might also be determined by the requirements of MCS-48 architecture (such as the restriction on certain jump instructions crossing page boundaries — see Chapter 3). This limitation could cause problems for macros containing such jumps, since you don't know when you call a macro whether it will straddle a page boundary after expansion. The command

```
        JC    ADDR1
```

generates an error if 'ADDR1' resides on a different page than the instruction itself. This specific problem might be solved by coding

```
        JNC  $+2
        JMP  ADDR1
```

since 'JMP' can cross boundaries. However, there may be similar situations that would warrant placing the 'ADDR1' code in a subroutine.

USING MACROS

The diskette-resident 8048/41 assembler recognizes the following macro operations:

- MACRO directive
- ENDM directive
- LOCAL directive
- REPT directive
- IRP directive
- IRPC directive
- EXITM directive
- Macro call

All of the directives listed above are related to macro definition. The macro call initiates the parameter substitution (macro expansion) process.

Macro Definition

Macros must be defined in your program before they can be used. A macro definition is initiated by the MACRO assembler directive, which lists the *name* by which the macro can later be called, and the *dummy parameters* to be replaced during macro expansion. The macro definition is terminated by the ENDM directive. The prototype instructions bounded by the MACRO and ENDM directives are called the *macro body*.

If label symbols appearing in a macro body have 'global' scope, multiply-defined symbol errors result if the macro is called more than once. A label can be given limited scope using the LOCAL directive. This directive causes a unique value to be assigned to the symbol by the assembler each time the macro is called and expanded. Dummy parameters also have limited scope.

Occasionally you may wish to duplicate a block of code several times, either within a macro or in line with other source code. This can be accomplished with minimal coding effort using the REPT (repeat block), IRP (indefinite repeat), and IRPC (indefinite repeat character) directives. Like the MACRO directive, these directives are terminated by ENDM.

The EXITM directive provides an alternate exit from a macro. When encountered, it terminates the current macro just as if ENDM had been encountered.

Macro Definition Directives

MACRO Directive

Label	*Opcode*	*Operand*
name	MACRO	dummy parameter(s)

The 'name' in the label field specifies the name of the macro body being defined. Any valid user-defined symbol name can be used as a macro name. Note that this name must be present and must *not* be terminated by a colon.

The optional dummy parameter can be any valid user-defined symbol name or may be null. If more than one parameter is listed, they are separated by commas. Dummy parameters have limited scope. If a reserved symbol is used as a dummy parameter, its reserved value is not recognized. It is considered strictly a dummy parameter limited to its specific macro definition. Dummy parameters are recognized in strings only when adjacent to the concatenation operator, '&' (described later). They are not recognized in comments.

Any MCS-48 or UPI-41 instruction or applicable assembler directive can be included in the macro body. The distinguishing feature of macro prototype text is that parts of it can be made variable by placing substitutable dummy parameters in instruction fields. These dummy parameters are the same as the symbols in the operand field of the MACRO directive.

Example: Define macro MAC1 with dummy parameters G1, G2, G3.

```
MAC1     MACRO   G1, G2, G3     ;MACRO DIRECTIVE
MOVES:   MOV     A,#G1          ;MACRO BODY
         MOV     R0,#G2
         MOV     R1,#G3
         ENDM                   ;ENDM DIRECTIVE
```

ENDM Directive

Label	*Opcode*	*Operand*
-----	**ENDM**	-----

The ENDM directive is required to terminate a macro definition and follows the last prototype instruction. It is also required to terminate code repetition blocks defined by the REPT, IRP, and IRPC directives. If the MACRO or code repetition directive is followed immediately by the ENDM directive, a null macro body results.

Any data appearing in the label or operand fields of an ENDM directive causes an error.

NOTE

> Because nested macro calls are not expanded during macro definition, the ENDM directive to close an outer macro cannot be contained in the expansion of an inner, 'nested' macro call. (See 'Nested Macro Definitions' later in this section.)

LOCAL Directive

Label	*Opcode*	*Operand*
-----	LOCAL	label name(s)

The specified symbolic label names are defined to have scope limited to the macro definition in which they are specified. Each time the macro is called and expanded, unique symbols of the form '??nnnn' are generated. Without this feature, multiple macro expansions would cause 'multiply-defined label' errors for each label in the macro body. The form '??nnnn' should not be followed for user-defined symbols. The first symbol is ??0001, the second ??0002, etc. The most recent symbol name generated always indicates the total number of symbols created for all macro expansions; these symbol names are never duplicated.

Operands specified as MACRO dummy parameters cannot be LOCAL directive operands in the same macro definition. Such operands would be recognized only as dummy parameters, and not as LOCAL operands.

Local symbols can only be defined within the macro body, and the LOCAL directive must precede the first line of prototype code. Any number of LOCAL directives can be specified, up to the limit of 255 total local symbols per macro.

A LOCAL directive appearing outside a macro definition causes an error. A name appearing in the label field of a LOCAL directive also causes an error.

Example:

```
MAC2      MACRO   G1, G2, G3
          LOCAL   MOVES
MOVES:    MOV     A,#G1
          MOV     R0,#G2
          MOV     R1,#G3
          ENDM
```

REPT Directive

Label	*Opcode*	*Operand*
optional:	REPT	expression

The REPT directive causes a sequence of source code lines to be repeated 'expression' times. 'Expression' may not include a forward reference. All lines appearing between the REPT directive and a subsequent ENDM directive constitute the block to be repeated.

The insertion of repeat blocks is performed in-line, when the assembler encounters the REPT directive. No explicit call is required to cause the code insertion since the definition is an implied call for expansion.

Example: Rotate accumulator right (through carry) six times.

```
ROTR6:  REPT  6
        RRC   A
        ENDM
```

The source code generated would be

```
        RRC A
        RRC A
        RRC A
        RRC A
        RRC A
        RRC A
```

IRP Directive

Label	*Opcode*	*Operand*
optional:	IRP	dummy param, ⟨list⟩

The IRP (indefinite repeat) directive creates a macro definition with one dummy parameter and is expanded once for each *actual parameter* supplied by 'list.' The definition is terminated by ENDM.

The list of actual parameters to be substituted for the dummy is separated by commas and enclosed in angle brackets (⟨ ⟩). Repetition continues until each element of the list has been substituted into the IRP macro body. A list of 'n' elements cause 'n' repetitions of the IRP body to be produced. If a null string is specified as the actual parameter, one iteration of the IRP body is produced with the null string substituted for each occurrence of the dummy parameter.

Note that, unlike MACRO, IRP supplies the actual parameters to be used as part of its prototype code definition (that is, the macro call is in-line with the macro definition). (See the discussion of 'Macro Calls' and 'Macro Expansion' later in this section.)

Example:

```
N1   EQU   1
N2   EQU   5
N3   EQU   7
      .
      .
      .
     IRP     X,<N1,N2,N3>
     ADD     A,#X
     ENDM
```

This example would generate the following code sequence:

```
     ADD     A,#1
     ADD     A,#5
     ADD     A,#7
```

IRPC Directive

Label	*Opcode*	*Operand*
optional:	IRPC	dummy param, text

The IRPC (indefinite repeat character) directive causes a sequence of macro prototype instructions to be repeated for each text character of the actual parameter specified, substituting the actual text character for each occurrence of the dummy parameter. If the text string is protected by optional angle brackets, any delimiter appearing in this text string will simply be substituted as text into the prototype code. A list of 'n' characters in the actual 'text' causes 'n' repetitions of the IRPC body to be produced. If a null string is specified as the actual parameter, one iteration of the IRP body is produced with the null string substituted for each occurrence of the dummy parameter.

Like IRP, IRPC creates a macro call in-line with the macro definition. It must also be terminated by ENDM.

Example:

```
CALSEQ: IRPC    X,01
        MOV     A,@R&X   ;;'&' CONCATENATES R AND X
        CALL    SUBR
        ENDM
```

This IRPC definition would generate the following code sequence:

```
     MOV  A,@R0
     CALL SUBR
     MOV  A,@R1
     CALL SUBR
```

Note that two special operators are used in this example: double semicolons and an ampersand. These and other operators are described in the subsection 'Special Operators,' below.

EXITM Directive

Label	*opcode*	*Operand*
optional:	EXITM	-----

EXITM provides an alternate way to terminate a macro expansion or REPT/IRP/IRPC repetition. When encountered in a macro body, it is equivalent to ENDM. Even though a macro body includes EXITM, however, it must still be terminated by an ENDM directive.

In the case of nested macro calls, EXITM causes an exit to the previous level of macro call expansion. In the case of REPT/IRP/IRPC expansions, EXITM terminates not only the current repetition, but all subsequent repetitions as well. The action following execution of EXITM is identical to that following completion of all repetitions.

Any data appearing in the operand field of an EXITM directive causes an error.

EXITM is commonly used as part of a conditional assembly as in the following example.

```
MAC3  MACRO   E,F
        .
        .
        .
      IF      E EQ 0
      EXITM
      ENDIF
        .
        .
        .
      ENDM
        .
        .
        .
      MAC3    0,1
```

The expansion of the above macro will be terminated when EXITM is assembled; that is, if 'E EQ 0' is TRUE.

Null Macros

A macro may legally consist of only the MACRO and ENDM directives. Thus, the following is a legal macro definition:

```
NADA    MACRO   P1,P2,P3,P4
        ENDM
```

A call to this macro produces no source code and therefore has no effect on the program.

Although there is no reason to write such a macro, the null (or empty) macro body has a practical application. For example, all the macro prototype instructions might be enclosed with IF-ENDIF conditional assembly directives. When none of the specified conditions is satisfied, all that remains of the macro is the MACRO directive and the ENDM directive.

Special Operators

In certain special cases, the normal rules for dealing with macros do not work. Assume, for example, that you want to specify three actual parameters and the second parameter happens to be the comma character. To the assembler, the list PARM1,,,PARM3 looks like a list of four parameters, with the second and third parameters missing. The list can be passed correctly by enclosing the comma in angle brackets: PARM1,<,>,PARM3. Angle brackets tell the assembler to accept the enclosed character(s) as actual parameter(s) rather than as delimiter(s).

The assembler recognizes several special operators in evaluating macro definitions. These are:

&　　Ampersand. Used to concatenate (link) text and dummy parameters. See the further discussion of ampersands below.

< >　　Angle brackets. Used to delimit text, such as lists, that contain other delimiters (including significant blanks). To pass such text to nested macro calls, use one set of angle brackets for each level of nesting. (See 'Nested Macro Definitions,' below.)

;;　　Double semicolon. Used before a comment in a macro definition to prevent inclusion of the comment in the definition and reduce storage requirements. The comment still appears in the listing of the definition.

!　　Exclamation point (escape character). Placed before a character (usually a delimiter) to be passed as literalized text in an actual parameter. Used primarily to pass angle brackets as part of an actual parameter. To pass a literalized exclamation point, issue '!!.' Carriage returns may not be passed as actual parameters.

NUL　　Logical null; returns a value of TRUE if the following operand is a null string.

When a macro is expanded, any ampersand preceding or following a dummy parameter in a macro definition is removed and the substitution of the actual parameter occurs at that point. When it is not adjacent to a dummy parameter, the ampersand is not removed and is passed as part of the macro expansion text.

If nested macro definitions (described below) contain ampersands, the only ampersands removed are those adjacent to dummy parameters belonging to the macro definition currently being expanded. All ampersands must be removed by the time the expansion of the encompassing macro body is performed. Exceptions force 'illegal character' errors.

Ampersands placed inside strings are recognized as concatenation delimiters when adjacent to dummy parameters; dummy parameters are recognized in strings only when adjacent to ampersands. Ampersands are not recognized as operators in comments.

Example:

```
MAC4   MACRO    D,E,F
D
       MOV      A,#E       ;;LOAD ACC
       ADD      A,R&F      ;;ADD REG CONTENTS
       MOV      R7,A       ;STORE RESULT IN REG 7
       ENDM
```

A subsequent call to macro MAC4, supplying actual parameters for dummy parameters 'D,E,F' as follows,

```
MAC4  <;THIS IS CALL1>  ,3AH,5
```

would cause the following substitution:

```
;THIS IS CALL 1
       MOV    A,#3AH
       ADD    A,R5
       MOV    R7,A         ;STORE RESULT IN REG 7
```

Example:

In a macro with the dummy parameters W, X, Y, Z it is acceptable for either X or Y to be null, but not both. The following IF directive tests for the error condition:

```
IF NUL X&Y
EXITM
```

Nested Macro Definitions

A macro definition can be contained completely within the body of another macro definition (that is, macro definitions can be *nested*). The body of a macro consists of all text (including nested macro definitions) bounded by matching MACRO and ENDM directives. The assembler imposes no limit on the depth of macro definition nesting.

Example:

```
LEVEL2  MACRO   H,J      \               \
        MOV     A,#H                     |
        ADD     A,#J                     |
        MOV     R4,A                     |
LEVEL1  MACRO   K,L      \               |
        MOV     A,#K     |   Level       |     Level
        ADDC    A,#L     >     1         >       2
        MOV     R3,A     |               |
        ENDM             /               |
        CLR     C                        |
        CLR     A                        |
        ENDM                             /
```

When a higher-level macro (LEVEL2 in this example) is called for expansion, the next lower-level macro (in this case LEVEL1) is defined and eligible to be called for expansion. A lower-level macro cannot be called unless all higher-level macro definitions have already been called and expanded.

A new macro may be defined or an existing macro redefined by a nested macro definition depending on whether the name of the nested macro is a new label or has previously been established as a dummy parameter in a higher-level macro definition. When dummy parameters are being identified in higher-level macros, all nested macro definitions are also scanned. Therefore, each time a higher-level macro is called, a lower-level definition can be defined differently if the two contain common dummy parameters. Such redefinition can be costly, however, in terms of assembler space used, particularly the availability of symbol table space, and execution speed.

Since IRP, IRPC, and REPT blocks constitute macro definitions, they also can be nested within another definition created by IRP, IRPC, REPT, or MACRO directives. In addition, an element in an IRP or IRPC actual parameter list (enclosed in angle brackets) may itself be a list of bracketed parameters; that is, lists of parameters can contain elements that are also lists.

Example:

```
LISTS   MACRO   PARAM1, PARAM2
          .
          .
          .
        ENDM
          .
          .
          .
        LISTS    <A, <B,C>>
```

Macro Calls

Once a macro has been defined, it can be called in a program any number of times. The call consists of the macro name and any actual parameters to replace dummy parameters during macro expansion. During assembly, each call encountered is replaced by the macro definition code with actual parameters substituted for dummy parameters.

Macro Call Format

Label	*Opcode*	*Operands*
optional:	macro name	actual parameter(s)

A macro must be defined before its first reference by a macro call. The macro body identified by 'macro name' is inserted into your program wherever the call appears. The specified actual parameters are substituted for the dummy parameters in the macro body.

Actual parameters must be specified in the same order as they are listed in the MACRO directive. If fewer parameters appear in the macro call than in the definition, a 'null' string is substituted for the remaining dummy parameters.

A null parameter can also be indicated by two consecutive commas or, in the case of the first parameter, by a single comma at the beginning of the operand field. If more actual parameters are specified than are listed in the definition, the extra parameters are ignored.

All blanks in an actual parameter list are considered to be delimiters and are not passed as part of the actual parameter. Angle brackets must enclose the actual parameter if blanks are to be passed literally (as in the case of any other delimiter passed as an actual parameter). Carriage returns may not be passed as parameters.

Example: Call MAC2 (defined earlier in our example of LOCAL directive usage). MAC2 was defined as:

```
MAC2     MACRO   G1,G2,G3
         LOCAL   MOVES
MOVES:   MOV     A,#G1
         MOV     R0,#G2
         MOV     R1,#G3
         ENDM
```

```
    Main Program         |     Substitution
                         |
CLR    C                 |          CLR    C
MAC2   0AH,0FFH,3AH      |??0001:   MOV    A,#0AH
ADD    A,R1              |          MOV    R0,#0FFH
MOV    R4,A              |          MOV    R1,#3AH
 .                       |          ADD    A,R1
 .                       |          MOV    R4,A
 .                       |           .
 .                       |           .
MAC2   0ACH,0FFH,HEXV    |           .
ANL    A,R0              |??0002:   MOV    A,#0ACH
 .                       |          MOV    R0,#0FFH
 .                       |          MOV    R1,#HEXV
 .                       |          ANL    A,R0
                         |
```

Nested Macro Calls

Macro calls can be nested within macro definitions up to eight levels (including any combination of nested IRP, IRPC, and REPT constructs). The body representing the nested macro call need not be defined when the macro containing the nested call is defined; however, it must be defined before the enclosing macro is called.

A macro definition can also contain nested calls to itself (*recursive macro calls*) up to eight levels, as long as the recursive macro expansions can be terminated eventually. This operation can be controlled using the conditional assembly directives described in Chapter 5 (IF, ELSE, ENDIF).

Example: Have a macro call itself a specific number of times after it is called from elsewhere in the program.

```
RECALL  MACRO
          .
          .
          .
        IF          PARAM1 NE 0
PARAM1  SET         PARAM1-1
        RECALL                      ;RECURSIVE CALL
        ENDIF
          .
          .
          .
        ENDM
```

If this macro is called with the sequence

```
PARAM1  SET     5
        RECALL
```

the macro will call itself five times.

Macro Expansion

When a macro is called, the actual parameters to be substituted into the prototype code can be passed in one of two modes. Normally, the substitution of actual parameters for dummy parameters is simply a *text* substitution. The parameters are not evaluated until the macro is expanded.

If a percent sign (%) precedes the actual parameter in the macro call, however, the parameter is evaluated immediately, before expansion occurs, and is passed as a decimal number.

Example:

```
X       SET     10
Y       SET     15
          .
          .
          .
MAC5    MACRO   L,M,N
Y       SET     0F0H
        MOV     A,#L
        ANL     A,#M
        MOV     R7,A
        ADD     A,#N
        ENDM
          .
          .
          .
        MAC5    %3H + Y/5H,Y,X
```

When the call to MAC5 is encountered, the text substitution is as follows:

```
Y  SET    0F0H
   MOV    A,#6
   ANL    A,#Y
   MOV    R7,A
   ADD    A,#X
```

'MOV A,#6' is the result of immediate evaluation using Y=15. 'ANL A,#Y' simply substitutes the text 'Y' for dummy parameter 'M.' Similarly, 'ADD A,#X' is the result of a simple text substitution.

The text expansion is as shown on the left:

```
                            Y SET 0F0H
00100011  00000110            MOV A,#6
01010011  11110000            ANL  A,#Y
10101111                      MOV R7,A
00000011  00001010            ADD A,#X
```

Note that when the expanion occurs, the value 'F0' (11110000) replaces 'Y.' This value is set by the first statement of the expansion. 'X' is replaced by '10,' its value by prior definition.

SAMPLE MACROS

The following samples further demonstrate the use of macro directives and operators.

Repetitive Addition (IRP)

The following example lets you add the contents of any number of data memory locations, leaving the result in the accumulator. The defined macro

```
ADDMEM      MACRO       LIST
            CLR         A
            IRP         SCR,⟨LIST⟩
            MOV         R0,#SRC
            ADD         A,@R0
            ENDM                            ;;END IRP BLOCK
            ENDM                            ;;END MACRO
```

when called with

```
ADDMEM      ⟨30,32,34⟩
```

produces the expansion

```
          CLR       A
          MOV       R0,#30
          ADD       A,@R0
          MOV       R0,#32
          ADD       A,@R0
          MOV       R0,#34
          ADD       A,@R0
```

The sum of the contents of bytes 30, 32, and 34 is left in the accumulator.

Repetitive Add and Store (IRPC, &)

In this example, a series of numbers is added to the accumulator and the subtotals stored in data memory. IRPC is used to reduce the coding required. The macro is defined as follows:

```
MOVTOT    MACRO       X,Y,Z
          IRPC        S,Z
          ADD         A,#X&&S
          MOV         R0,#Y&&S
          MOV         @R0,A
          ENDM
          ENDM
```

The call

```
MOVTOT    SRC,TOTAL,123
```

produces the expansion

```
ADD         A,#SRC1
MOV         R0,#TOTAL1
MOV         @R0,A
ADD         A,#SRC2
MOV         R0,#TOTAL2
MOV         @R0,A
ADD         A,#SRC3
MOV         R0,#TOTAL3
MOV         @R0,A
```

Multiplication (REPT,LOCAL)

This example uses REPT to perform the 8-bit multiplication shown in the example on page 3-47. As in that example, two 8-bit numbers are multiplied and their 16-bit product stored in registers 2 and 3. REPT replaces the loop mechanism of the earlier example, which generates more code but executes more quickly.

```
FST8X8: MOV     R6,#MCAND         ;MULTIPLICAND IN REG 6
        MOV     R3,#MPLIER        ;MULTIPLIER, LOW PARTIAL
                                  ;PRODUCT IN REG 3
        CLR     A
        CLR     C
        REPT    9                 ;;BEGIN REPEAT BLOCK
        LOCAL   NOADD
        RRC     A                 ;;ROTATE
        XCH     A,R3              ;; CARRY, ACC, REG 3
        RRC     A                 ;;     RIGHT
        XCH     A,R3              ;;        ONE BIT
        JNC     NOADD             ;;TEST CARRY
        ADD     A,R6
NOADD:
        ENDM                      ;;END REPEAT BLOCK
        MOV     R2,A              ;STORE HIGH PARTIAL
                                  ;  PRODUCT
```

Zero and Label Contiguous Locations (REPT, &, %)

In this example, the REPT directive is used to zero and label each location in a defined data block. Two macros are defined:

- INCR generates labels and DB directives for each location to be zeroed.
- BLOCK specifies the number of locations to be zeroed (NUMB) and supplies the label prefix (PREFIX) and suffix (CNT) to INCR.

Note that assembler controls (lines beginning with '$') are embedded in the macro definition code. These are discussed in more detail in Chapter 8. Generally, the controls specified here reduce the size of the assembly listing; the expansion for INCR is shown in the listing, but the expansion of BLOCK is suppressed.

```
;DEFINITION OF INCR
INCR     MACRO     F1,F2
$SAVE GEN
F1&F2:   DB        0
$RESTORE
         ENDM

;DEFINITION OF BLOCK
BLOCK    MACRO     NUMB,PREFIX,CNT
$SAVE    NOGEN
COUNT    SET       CNT
         REPT      NUMB

COUNT    SET       COUNT+1
         INCR      PREFIX,%COUNT
         ENDM
$RESTORE
         ENDM
```

The macro call

```
        BLOCK           3,LOC,40
```

produces the listing

```
                BLOCK   3,LOC,40
        LOC40:  DB      0
        LOC41:  DB      0
        LOC42:  DB      0
```

Without the assembler controls, the listing would be

```
                BLOCK   3,LOC,40
        COUNT   SET     40
                REPT    3
        COUNT   SET     COUNT+1
                INCR    LOC,%COUNT
                ENDM
        COUNT   SET     COUNT+1
                INCR    LOC,%COUNT
        LOC40:  DB      0
        COUNT   SET     COUNT+1
                INCR    LOC,%COUNT
        LOC41:  DB      0
        COUNT   SET     COUNT+1
                INCR    LOC,%COUNT
        LOC42:  DB      0
```

Altering Macro Expansions (Three Approaches)

This example uses conditional assembly, the EXITM directive, and a nested macro definition to provide three approaches to a problem. The problem is to define a macro so that identical calls produce different expansions.

Our macro (SBMAC) needs a subroutine (SUBR) to perform its function; space constraints rule against the extra bytes generated by repetitive in-line substitution. We would like SBMAC to perform an in-line substitution the first time it is called, then call SUBR for each subsequent macro call.

Note in these examples that the label SUBR cannot be declared LOCAL, as it must be called from outside the first expansion (that is, it must be global). This use of a global label in a macro is possible only when that part of the macro body containing the label is expanded only once.

First Solution (Conditional Assembly)

The first solution uses the setting of a switch (FIRST) and the conditional assembly directives. The switch is set TRUE and the macro defined as follows.

```
TRUE        EQU         0FFH
FALSE       EQU         0
FIRST       SET         TRUE
SBMAC       MACRO
            CALL        SUBR
$SAVE       NOCOND
            IF          FIRST
FIRST       SET         FALSE
            JMP         NEXT
SUBR:        .
             .
             .
             .
            RET
NEXT:
            ENDIF
$RESTORE
            ENDM
```

The first call to SBMAC expands the full definition, including the call to and definition of SUBR. The assembler control NOCOND suppresses the listing of conditional assembly directives and conditionally-assembled code.

```
            SBMAC
            CALL        SUBR
FIRST       SET         FALSE
            JMP         NEXT
SUBR:        .
             .
             .
             .
            RET
NEXT:
```

Because FIRST is TRUE when encountered during this expansion, the statements between IF and ENDIF are assembled into the program. The statement following IF sets FIRST to FALSE. In subsequent calls, the conditionally-assembled code is skipped and the subroutine is not regenerated. Only the following expansion is produced.

```
            SBMAC
            CALL        SUBR
```

Second Solution (Conditional Assembly, EXITM)

This solution closely parallels the first, except that EXITM is used to terminate the unnecessary expansion after the first call. EXITM is assembled only when FIRST is FALSE, which it is after the first call to SBMAC.

```
TRUE       EQU       0FFH
FALSE      EQU       0
FIRST      SET       TRUE
SBMAC      MACRO
           CALL      SUBR
$SAVE      NOCOND
           IF        NOT FIRST
           EXITM
           ENDIF
FIRST      SET       FALSE
           JMP       NEXT
SUBR:       .
            .
            .
            .
           RET
NEXT:
$RESTORE
           ENDM
```

The expansion is the same as for the first solution.

Third Solution (Nested Macro)

This solution uses a nested macro to redefine a higher-level macro, so that the higher-level macro is not expanded after the first call.

```
SBMAC      MACRO
SBMAC      MACRO
           CALL      SUBR
           ENDM
           CALL      SUBR
           JMP       NEXT
SUBR:       .
            .
            .
            .
           RET
NEXT:
           ENDM
```

The first call to SBMAC expands the higher-level definition containing the subroutine definition and call. It also redefines the macro to be simply a subroutine call.

```
          SBMAC
SBMAC     MACRO
          CALL      SUBR
          ENDM
          CALL      SUBR
          JMP       NEXT
SUBR:      .
           .
           .
           .
          RET
NEXT:
```

Subsequent calls to SBMAC expand the subroutine call only.

```
          SBMAC
          CALL      SUBR
```

PART TWO

ASSEMBLER OPERATION

7. Assembler Overview

8. Assembler Controls

9. Assembler Operation

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

7. ASSEMBLER OVERVIEW

An assembler performs the clerical function of converting your assembly language program into machine-executable form. It accepts your source file and, depending on the output options selected, produces an executable object file, a listing of the source and assembled code, and a symbol cross-reference listing.

VERSIONS OF ASSEMBLER

The MCS-48 and UPI-41 assemblers are available in three versions:

1. The 'Intellec MONITOR MCS-48/UPI-41 Assembler' runs under control of the monitor on the Intellec Microcomputer Development System and is delivered in paper tape form.

2. The 'Series II MCS-48/UPI-41 ROM Assembler' runs under control of the monitor on the Intellec Series II Model 210 and resides in ROM.

3. If your Intellec configuration includes a diskette unit, you can use the 'ISIS-II MCS-48/UPI-41 Macro Assembler' (ASM48). This assembler runs under the Intel Systems Implementation Supervisor (ISIS-II).

Details for loading and controlling the MONITOR and ISIS-II assemblers are given in Chapters 8 and 9. Error messages issued by the assemblers are listed in Appendix F. The hardware/software environment requirements are summarized below. Operation of the ROM assembler is described in the document *Intellec Series II Model 210 User's Guide* (9800557).

MONITOR Assembler Environment

The paper-tape-resident assembler uses the following hardware:

- Intellec system with 16K RAM memory
- Console device (TTY or CRT)
- Paper tape reader/punch
- Line printer (if available)

The Intellec monitor package is the only required software.

ISIS-II Assembler Environment

The diskette-resident assembler uses the following hardware:

- Intellec MDS-800 or Intellec Series II system with 32K RAM memory (48K if source contains macros)
- Console device (TTY or CRT for MDS-800; built-in with Intellec Series II)
- One or more diskette drives
- Line printer (if available)

This assembler also requires the ISIS-II software package.

INPUT/OUTPUT FILES

Source File

The input to the assemblers is a source file, which can contain three elements:

- An assembly language program, composed of instructions described in Chapters 3 and 4;
- Assembler directives, described in Chapters 5 and 6;
- Assembler control lines, described in Chapter 8

Object File

The MONITOR assembler produces an object file on the paper tape punch unit. The ISIS-II assembler can output its object file to any file or output device recognized by ISIS.

The object file contains machine language instructions and data that can be loaded into memory for execution or interpretation. In addition, it contains control information governing the loading process (such as the starting address for program execution). An object file can also be used to program an MCS-48 or UPI-41 ROM or EPROM device.

Both assemblers produce object files in hexadecimal format. This format and special records generated for paper tape object files are described in the document *Object File Formats, An Intel Software Standard* (98-183).

List File

The list file is a formatted file designed to be output to a line printer or terminal, but it can be sent to any file or output device under ISIS-II. It includes listings of:

- Your assembled object code;
- Your source program;
- A table of symbols and their values;
- A summary of assembly errors.

The formats of these list file components are described in Appendix D.

Symbol-Cross-Reference File

During the first pass of both assemblers, a file of symbol-cross-reference records is created, if requested. This file is punched into paper tape by the Intellec MONITOR assembler, or written to a diskette file named ASXREF.TMP by the ISIS-II assembler.

In general, the assemblers generate two types of symbol-cross-reference records: *symbol-definition records* and *symbol-reference records.* If a symbol appears as a name in a label field and the symbol is being defined (by SET, EQU, or MACRO, or as a label), a symbol-definition record is produced. If the symbol is being redefined (by SET or MACRO), it is considered a symbol definition. All other symbol occurrences are considered references and cause the assembler to generate a symbol-reference record each time the symbol appears. Symbol definition records are terminated by a pound sign (#) in the cross-reference listing.

All symbols are cross referenced except dummy parameters and local labels appearing in macro definitions (that is, all global user-defined symbols, macro names, and actual symbols replacing dummy parameters or local labels are cross referenced).

A listing of the cross-reference file can be produced by reading it into the assembler cross-reference generator. In the paper tape environment, this program is loaded after the assembly and run with the assembler-generated cross-reference file as input. In the diskette environment, the assembler calls on ISIS-II to load the generator program (ASXREF) and cross-reference file (ASXREF.TMP) from the diskette. From the programmer's point of view, these ISIS-II operations are automatic (once the cross-reference file has been requested). The format of the cross-reference listing is shown in Appendix D.

ISIS-II Assembler Reserved File Names

The ISIS-II assembler uses several files of its own, such as the intermediate cross-reference file just mentioned. While you don't need to remember the names of these files, you must know where they reside to avoid diskette space conflict.

The assembler root program (ASM48) and its overlays (ASM48.OVn, where n = 0, 1, 2, ...) must reside on the same diskette, but this diskette can be on any drive. The cross-reference generator (ASXREF) must reside on this diskette also.

The intermediate cross-reference file (ASXREF.TMP) is written to the drive containing your source file. The MACROFILE control determines where the intermediate macro file (ASMAC.TMP) is written; the default is the source file drive.

8. ASSEMBLER CONTROLS

INTRODUCTION TO ASSEMBLER CONTROLS

Assembler controls allow you to specify the input/output files or devices to be used by the assembler and whether list or object files (or portions of these files) are to be generated by the assembler.

For both the MONITOR and ISIS-II assemblers, these controls can be specified at two levels:

- In commands specified at assembly time
- As control lines embedded throughout your source file

The latter allow selective control over sections of your program. For example, you might want to suppress the assembly listing for certain sections of your program, or to cause page ejects at specific places.

Primary and General Controls

Controls are classified as *primary* and *general.* The interpretation of these terms differs somewhat between the ISIS-II and MONITOR assemblers because of the different ways you can interface with these two assemblers. The ISIS-II assembler runs without interruption once it is called; the MONITOR assembler may require several passes, with additional controls specified at the beginning of each pass.

Both classes of controls can be set when the assembler is run or in source file control lines. However, source file control lines containing primary controls must be inserted before the first line of comments or source code. General controls can be respecified at any time.

The ISIS-II assembler allows primary controls to be specified only once. This applies to controls specified in assembly-time command lines, to control lines embedded in your source code, or combinations of the two.

When the MONITOR assembler is run, primary controls can be specified *or respecified* in the command preceding each pass. A primary setting in a source code control line cannot change a primary control set in any previous command or control line. The precedence of *primary controls* in the MONITOR assembler is:

1. Current pass command
2. Previous pass command
3. Current control lines
4. Default settings

Specifying Controls

Controls can be specified using either upper-case or lower-case characters.

If a control is specified incorrectly in an assembly-time command, the entire command is ignored and must be re-entered.

If a control is specified incorrectly in a source code control line, the incorrect control and all controls following it in the line are ignored.

Summary of Controls

The following list shows the controls available, their basic functions, whether they are recognized by *both* assemblers or *ISIS-II only* (B/I), and whether they are *primary* or *general* (P/G). Default controls are italicized. The remainder of this chapter describes each control in greater detail.

Control	*B/I*	*P/G*	*Function Area*
OBJECT/NOOBJECT	I	P	Object File
DEBUG/*NODEBUG*	B	P	Object File
PRINT/NOPRINT	I	P	Assembly Listing
COND/NOCOND	B	G	Assembly Listing
LIST/NOLIST	B	G	Assembly Listing
SYMBOLS/NOSYMBOLS	B	P	Assembly Listing
XREF/*NOXREF*	B	P	Cross-Reference Listing
PAGING/NOPAGING	B	P	Listing Format
PAGELENGTH *(66)*	B	P	Listing Format
PAGEWIDTH *(120)*	B	P	Listing Format
TITLE	B	G	Listing Format
EJECT	B	G	Listing Format
GEN/NOGEN	I	G	Macro List
MACRODEBUG/*NOMACRODEBUG*	I	G	Macro List/Object Files
MACROFILE/*NOMACROFILE*	I	P	Macro Operation
MOD21	B	P	8021
MOD41	B	P	UPI-41
SAVE	I	G	Stack Controls
RESTORE	I	G	Fetch Controls
INCLUDE	I	G	Library Function

ISIS-II ASSEMBLER CONTROLS

ISIS-II Assembly-Time Command

The ISIS-II MCS-48/UPI-41 Macro Assembler is invoked by calling the ISIS-II file ASM48. This call includes the name of your source file and any assembler controls you wish to specify. Items in the control list are separated by spaces. The call is terminated by a carriage return.

```
—[:Fn:] ASM48 file control-list
```

The 'file' in this format is your source file. This file (and files enclosed in parentheses as part of a control) can be a 1-6 character file name, a file name followed by a period and 1-3 character extension, an ISIS-II device name, or an

ISIS-II device name followed by a file name and extension. (See the *ISIS-II System Users' Guide* for details.)

Examples:

FILE20	(filename)
PROG.SRC	(filename.extension)
:HR:	(:ISIS-II device name:)
:F1:ASSMB. SRC	(:ISIS-II dev name:filename.ext)

All control items specified must be spelled out in their entirety. If no diskette file is specified preceding 'ASM48', ':F0:' is assumed.

Example:

```
—ASM48    PROG.SRC    DEBUG    SYMBOLS    XREF
```

Primary Controls

Control	*Effect*
OBJECT(file)	An object code file is generated and is output to the specified device. If this control is omitted, 'OBJECT (file.HEX)' is assumed, where 'file' is the name of your source file.
NOOBJECT	Object code generation is suppressed.
DEBUG	If an object file is requested, the symbol table is output to that file. DEBUG has no effect otherwise.
NODEBUG	The symbol table is not included in the object file.
PRINT(file)	An assembly list file is opened and is output to the specified intermediate file. If this control is omitted, 'PRINT(file.LST)' is assumed, where 'file' is the name of your source code file. See general control LIST.
NOPRINT	The assembly output listing is suppressed. No file is specified for listing; therefore, no listing output is possible.
SYMBOLS	If a list file is opened by PRINT, the symbol table is output to the list file. SYMBOLS has no effect otherwise.
NOSYMBOLS	The symbol table is not included in the list file created by PRINT.
XREF	A symbol-cross-reference file is requested. An intermediate file is output to ASXREF.TMP and the cross-reference listing to the file created by PRINT.

Control *Effect*

NOXREF

Symbol-cross-reference file generation is suppressed.

MACROFILE(drive)

All macro definition files are directed to the specified drive. If no drive is specified, the drive where the source file resides is used. Your Intellec system must have at least a 48K memory if MACROFILE is specified.

NOMACROFILE

No macro temporary files are created. If your source file contains macros, all definitions and calls cause errors. This control allows the assembler to run on a 32K-memory Intellec system.

PAGELENGTH(n)

Each list file page is 'n' lines long, where 'n' must be at least 15 and includes 3 blank lines at the top of the page, 3 blank lines at the bottom of the page, and any page headings specified. If 'n' is ≤ 14, PAGELENGTH is set to 15. The default value is 66.

Note that 3 blank lines are issued to reach the next 'top-of-page' as opposed to issuing form feeds to reach the physical 'top-of-form.'

PAGEWIDTH(n)

Each list file line can be up to 'n' characters long, where 'n' must be in the range 72≤n≤132. Lines exceeding the page width are continued in column 25 of the following line (but lines >132 characters are truncated to 132). The default page width is 120.

PAGING

Assembler separates listing into pages with headers at each page break.

NOPAGING

Listing is not separated into pages. Headers are printed only once, at the beginning of the listing.

MOD21

Assembler assumes 8048 code is being assembled unless the 8021 instruction set is specified by this control. A warning is issued if an instruction not recognized by the 8021 is specified while this control is set, or if an instruction unique to the 8021 is specified without setting this control.

MOD41

Assembler assumes 8048 code is being assembled unless the UPI-41 instruction set is specified by this control. A warning is issued if an instruction unique to the 8041 is specified without setting this control, or if an instruction not recognized by the 8041 is issued while this control is set.

General Controls

Control	*Effect*
INCLUDE(file)	Subsequent source lines are input from specified file until an end-of-file or nested INCLUDE is found. (Nesting may be four deep.) Following the end-of-file, input resumes from the file being processed when the INCLUDE was encountered.
LIST	An assembly output listing is generated and sent to the file specified by PRINT.
NOLIST	Assembly listing is suppressed, except header, symbol table, cross-reference table, and lines containing errors.
COND	Conditionally-skipped source code is included in the assembly listing if LIST is selected. The conditional-assembly directives are also listed.
NOCOND	Listing of conditionally-skipped source code and conditional-assembly directives is suppressed. Listing of the EXITM directive is suppressed also.
MACRODEBUG	Assembler-generated macro symbols are output to the the list and object files when the symbol table is output.
NOMACRODEBUG	Assembler-generated macro symbols are not output to the list and object files.
GEN	Macro expansion source text generated by macro calls is listed if LIST is selected.
NOGEN	Macro expansion source text listing is suppressed.
TITLE('string')	The specified string is printed in character positions 1-64 of the second line of a page header. Strings longer than 64 characters are truncated. 'String' cannot be null. TITLE remains in effect until another TITLE is encountered. A blank line results if TITLE is not specified.
EJECT	Spaces are skipped to the next top-of-form. The position of the next top-of-form is determined by PAGELENGTH, not by the physical top-of-form.
SAVE	Current settings of LIST, COND, and GEN controls are stacked (but remain valid until explicitly changed). Controls can be stacked up to eight levels deep.
RESTORE	Control settings at the top of the stack are restored.

Defaults

The following defaults are assumed by the ISIS-II assembler if the corresponding controls are not selected.

```
OBJECT(file.HEX)
NODEBUG
PRINT(file.LST)
LIST
SYMBOLS
COND
GEN
NOXREF
NOMACRODEBUG
NOMACROFILE
PAGING
PAGELENGTH(66)
PAGEWIDTH(120)
```

ISIS-II Embedded Control Lines

The format for control lines embedded in source files to be processed by the ISIS-II assembler is

```
$control list
```

where '$' must appear in column 1 and items in the control list are separated by spaces.

Example:

```
$LIST    DEBUG    XREF    MACRODEBUG
```

Control lines containing primary controls must appear before the first statement in the source file, including comments. Control lines containing only general controls can be interspersed throughout the source file.

A control line containing more than one control is scanned from left to right. If a control is specified incorrectly, it is ignored, as are all remaining controls on that line.

The specific controls available and the defaults for unspecified controls are the same as described above in 'ISIS-II Assembly-Time Command.'

INTELLEC MONITOR ASSEMBLER CONTROLS

MONITOR Assembly-Time Commands

When the MONITOR assembler is loaded and goes into execution, it prompts with

```
P=
```

At this point, the assembler is asking you to specify the 'pass number' and controls you want, in the format:

```
passno control-list
```

The possible 'passno' options are:

1	Build symbol table.
2	Generate assembly listing.
3	Punch object file on paper tape.
4	Generate both assembly listing and object file. This optionshould be used only if the list and object files are assigned to different devices via the Intellec Monitor I/O ASSIGN command.
E	Exit assembler. Return control to Intellec monitor.

Pass 1 must be executed first. Any pass may then be executed in any sequence.

A new source tape can be assembled without reloading the assembler by issuing the monitor command '.G20' after the exit command (P=E). This action resets all options to their default values, thus allowing new options to be specified for the next assembly.

Primary Controls

Control	*Effect*
DEBUG	The symbol table is output to the object code file when pass 3 or pass 4 is executed.
NODEBUG	The symbol table is not included in the object code file.
SYMBOLS	The symbol table is included in the assembly listing when pass 2 or pass 4 is executed.
NOSYMBOLS	Symbol table listing is suppressed.
XREF	A symbol-cross-reference file is generated and output to paper tape during pass 1. To have any effect, XREF must be specified in your source file or when pass 1 is elected.

Control — *Effect*

NOXREF

Symbol-cross-reference output is suppressed.

PAGELENGTH(n)

Each list file page is 'n' lines long, where 'n' must be at least 12 and includes 3 blank lines at the top of the page, 3 blank lines at the bottom of the page, and any page headings specified. If 'n' is < 11, PAGELENGTH is set to 12. The default value is 66.

Note that 3 blank lines are issued to reach the next 'top-of-page' as opposed to issuing form feeds to reach the physical 'top-of-form.'

PAGEWIDTH(n)

Each list file line can be up to 'n' characters long, where 'n' must be in the range $72 \leq n \leq 132$. Lines exceeding the page width are continued in column 25 of the following line (but lines >132 characters are truncated to 132). The default page width is 120.

PAGING

Assembler separates listings into pages with headers at each page break.

NOPAGING

Listing is not separated into pages. Headers are printed only once, at the beginning of the listing.

MOD21

Assembler assumes 8048 code is being assembled unless the 8021 instruction set is specified by this control. A warning is issued if an instruction not recognized by the 8021 is specified while this control is set, or if an instruction unique to the 8021 is issued without setting this control.

MOD41

Assembler assumes 8048 code is being assembled unless the UPI-41 instruction set is specified by this control. A warning is issued if an instruction unique to the 8041 is specified without setting this control, or if an instruction not recognized by the 8041 is issued while this control is set.

General Controls

Control — *Effect*

LIST

Enables the assembly listing requested by specifying pass 2 or pass 4.

NOLIST

Disables the assembly listing requested by specifying pass 2 or pass 4, except for header, symbol table, cross-reference table, and lines containing errors.

COND

Conditionally-skipped source code is included in pass 2 or pass 4 listing if LIST is selected. The conditional-assembly directives are also listed.

Control	*Effect*
NOCOND	Listing of conditionally-skipped source code and conditional-assembly directives is suppressed.
TITLE('string')	The specified 'string' is printed in character positions 1-64 of the second line of the page header. Strings longer than 64 characters are truncated. 'String' cannot be null. TITLE remains in effect until another TITLE is encountered. A blank results if TITLE is not specified.
EJECT	Spaces are skipped to the next top-of-form. The position of the next top-of-form is determined by PAGELENGTH, not by the physical top-of-form.

Defaults

The following defaults are assumed by the MONITOR assembler if the corresponding controls are not selected.

NODEBUG
LIST
SYMBOLS
COND
NOXREF
PAGING
PAGELENGTH(66)
PAGEWIDTH(120)

MONITOR Embedded Control Lines

The format for control lines embedded in source files to be processed by the MONITOR assembler is

```
$control list
```

where '$' must appear in column 1 and items in the control list are separated by spaces.

Example:

```
$LIST    DEBUG    SYMBOLS    NOXREF
```

Control lines containing primary controls must appear before the first statement in the source file, including comments. Control lines containing only general controls can be interspersed throughout the source file.

A control line containing more than one control is scanned from left to right. If a control is specified incorrectly, it is ignored, as are all remaining controls on that line.

The specific controls available and the defaults for unspecified controls are the same as described above in 'MONITOR Assembly-Time Commands.'

9. ASSEMBLER OPERATION

The ISIS-II MCS-48/UPI-41 Macro Assembler is loaded by calling ASM48 at the ISIS-II command level and specifying your source file along with any desired assembler controls (Chapter 8). All assembler operations requested are performed without further intervention once the assembler begins execution.

The MONITOR MCS-48/UPI-41 Assembler must be loaded from paper tape. In addition, all peripheral device assignments must be made before the assembler begins execution. See the Intellec operator's manual for details.

ISIS-II ASSEMBLER OPERATION

Activation Sequence

The following example sequence activates and completes an ISIS-II assembly.

```
[:Fn:]ASM48    PROG.SRC    SYMBOLS    NODEBUG
```

Following the ISIS-II command prompt (-), a command to assemble the file PROG.SRC is issued. An assembly listing and object code file are requested and will be output by default to PROG.LST and PROG.HEX respectively. In addition, a symbol table listing will be performed, but symbol table output to the object file will be suppressed. Note that the same effect can be achieved with no controls specified, since the controls specified are both defaults.

```
ISIS-II MCS-48/UPI-41 MACRO ASSEMBLER, V1.0
```

The assembler sends out its sign-on message to the console device.

```
ASSEMBLY COMPLETE, NO ERRORS
```

After executing all assembler passes and completing the requested assembly listing and object code output, the assembler issues a sign-off message and error summary. If XREF is selected, the sign-on message

```
ISIS-II ASSEMBLER SYMBOL CROSS REFERENCE V1.0
```

is then issued on the console.

Sample Assembly

The following example illustrates normal use of the ISIS-II assembler. A short program (MADD.SRC) is taken through all the steps needed to activate the assembler and obtain an object code file and assembly and symbol-cross-reference listings. The source program to be assembled is shown first, followed by the assembler activation sequence. The resulting assembly and symbol-cross-reference listings are also shown.

The source code for program MADD.SRC follows:

```
;DECIMAL ADDITION ROUTINE. ADD BCD NUMBER
;AT LOCATION 'BETA' TO BCD NUMBER AT 'ALPHA' WITH
;RESULT IN 'ALPHA.' LENGTH OF NUMBER IS 'COUNT' DIGIT
;PAIRS. (ASSUME BOTH BETA AND ALPHA ARE SAME LENGTH
;AND HAVE EVEN NUMBER OF DIGITS OR MSD IS 0 IF
;ODD)
INIT    MACRO   AUGND, ADDND, CNT
        MOV     R0,#AUGND
L1:     MOV     R1,#ADDND
        MOV     R2,#CNT
        ENDM
;
ALPHA   EQU     30
BETA    EQU     40
COUNT   EQU     5
        ORG     100H
        INIT    ALPHA,BETA,COUNT
        CLR     C
LP:     MOV     A,@R0
        ADDC    A,@R1
        DA      A
        MOV     @R0,A
        INC     R0
        INC     R1
        DJNZ    R2,LP
        END
```

The ISIS-II assembler performs its operations without further user intervention after it is loaded. In this example, both assembly listing and object output are requested by default. The sample program is assumed to be on the system diskette with the name MADD.SRC. The activation sequence proceeds as follows:

```
-ASM48   MADD.SRC   SYMBOLS   XREF   MACROFILE
```

The source input file is specified as MADD.SRC. The PRINT control is selected and defaulted to file MADD.LST. The OBJECT control is also selected and defaulted to file MADD.HEX. Symbol table output to the list file is requested as well as a symbol-cross-reference listing. MACROFILE must be specified since the program contains a macro.

The assembly and cross-reference listings are shown below. For a detailed explanation of each item in these listings, see Appendix D.

```
—ASM48   MADD.SRC   SYMBOLS   XREF   MACROFILE
```

```
ISIS-II MCS-48/UPI-41 MACRO ASSEMBLER, V1.0                    PAGE 1

LOC   OBJ      SEQ                SOURCE STATEMENT

                 1  ;DECIMAL ADDITION ROUTINE. ADD BCD NUMBER
                 2  ;AT LOCATION 'BETA' TO BCD NUMBER AT 'ALPHA' WITH
                 3  ;RESULT IN 'ALPHA.' LENGTH OF NUMBER IS 'COUNT' DIGIT
                 4  ;PAIRS. (ASSUME BOTH BETA AND ALPHA ARE SAME LENGTH
                 5  ;AND HAVE EVEN NUMBER OF DIGITS OR MSD IS 0 IF
                 6  ;ODD)
                 7  INIT        MACRO      AUGND,ADDND,CNT
                 8              MOV        R0,#AUGND
                 9  L1:         MOV        R1,#ADDND
                10              MOV        R2,#CNT
                11              ENDM
                12  ;
001E            13  ALPHA       EQU        30
0028            14  BETA        EQU        40
0005            15  COUNT       EQU        5
0100            16              ORG        100H
                17              INIT       ALPHA,BETA,COUNT
0100  B81E      18 +            MOV        R0,#ALPHA
0102  B928      19 + L1:        MOV        R1,#BETA
0104  BA05      20 +            MOV        R2,#COUNT
0106  97        21              CLR        C
0107  F0        22  LP:         MOV        A,@R0
0108  71        23              ADDC       A,@R1
0109  57        24              DA         A
010A  A0        25              MOV        @R0,A
010B  18        26              INC        R0
010C  19        27              INC        R1
010D  EA07      28              DJNZ       R2,LP
                                END

USER SYMBOLS

ALPHA    001E    BETA     0028    COUNT    0005    LP      0107

L1       0102

ASSEMBLY COMPLETE, NO ERRORS
```

The 'ASSEMBLY COMPLETE' message is also issued on the console, followed by the cross-reference sign-on message if a cross-reference listing has been requested.

```
ISIS-II ASSEMBLER SYMBOL CROSS REFERENCE, V1.0                    PAGE 1

SYMBOL CROSS REFERENCE

ALPHA           13#     17      18
BETA            14#     17      19
COUNT           15#     17      20
INIT             7#     17
L1              19#
LP              22#     28
```

Listing is complete, sign-off message is issued on the listing, followed by ISIS prompt.

```
CROSS REFERENCE COMPLETE
```

Note that the NOLIST, NOOBJECT controls could have been specified to request just the error summary on the console, and a listing of the lines containing errors.

INTELLEC MONITOR ASSEMBLER OPERATION

Activation Sequence

The following example sequence activates and completes an assembly using the MONITOR assembler. Note that the paper tape assembler is delivered in two parts, either of which may be loaded first. Assembler console output is italicized in the following sequence.

1. Load half the assembler onto the paper tape reader and issue the Intellec monitor command

   ```
   .R0
   ```

   This reads half the assembler into Intellec memory.

2. Load the rest of the assembler onto the paper tape reader and again issue the monitor command

   ```
   .R0
   ```

   The entire assembler now resides in Intellec memory.

3. Load the paper tape containing your source program onto the paper tape reader.

4. Issue the monitor command

   ```
   .G20
   ```

   This initiates assembler execution. The assembler responds by sending a sign-on message to the console device.

```
INTELLEC MONITOR MCS-48/UPI-41 ASSEMBLER V1.0
```

It then prompts with

```
P=
```

5. Enter the pass number. This must be '1' the first time you respond. The assembler reissues the 'P=' prompt at the end of each pass, to which you can respond with any pass option.

   ```
   P=1 XREF
   ```

   In this example, pass 1 is specified and a symbol-cross-reference file requested.

6.
   ```
   PASS 1 COMPLETE
   P=2 NOCOND
   ```

   After pass 1, the assembler issues a completion message on the console output device and requests the next pass number and controls. The source program paper tape must be reloaded before specifying the next pass. In this example, pass 2 is specified and the NOCOND control selected.

7.
   ```
   PASS 2 COMPLETE. NO ERRORS
   P=3 NODEBUG
   ```

   Following passes 2, 3, and 4, an error summary and completion message are issued. The assembler then prompts for the next pass number and controls. The source paper tape must be reloaded again before specifying the next pass. In this example, pass 3 is specified and the NODEBUG control selected.

8.
   ```
   PASS 3 COMPLETE. NO ERRORS
   P=E
   ```

   In this example, the exit command is specified and control returns to the monitor. The cross-reference-generator paper tape is now loaded on the paper tape reader (to print out the cross-reference file requested in pass 1).

9.
   ```
   .R0
   ```

   This command to the Intellec monitor reads the cross-reference-generator program into Intellec memory. Next load the paper tape created during pass 1. This tape contains the cross-reference file.

10.
    ```
    .G20
    ```

    This monitor command initiates cross-reference-generator execution.

11. The generator program issues a sign-on message on the console output device.

    ```
    INTELLEC MONITOR ASSEMBLER SYMBOL CROSS REFERENCE, V1.0
    ```

    Press the carriage return key at this point. The cross-reference file is read and the listing generated. The generator program signs off on the listing device when finished.

```
CROSS REFERENCE COMPLETE
```

Control returns to the Intellec monitor, which then prompts for a new command.

Sample Assembly

The following example illustrates normal use of the MONITOR assembler. A short program is taken through all the steps needed to activate the assembly and cross-reference listings. The source program to be assembled is shown first, followed by the necessary passes through the assembler. The resulting assembly and symbol-cross-reference listings are also shown.

The following is the sample source program to be assembled.

```
;DECIMAL ADDITION ROUTINE. ADD BCD NUMBER
;AT LOCATION 'BETA' TO BCD NUMBER AT 'ALPHA' WITH
;RESULT IN 'ALPHA.' LENGTH OF NUMBER IS 'COUNT' DIGIT
;PAIRS. (ASSUME BOTH BETA AND ALPHA ARE SAME LENGTH
;AND HAVE EVEN NUMBER OF DIGITS OR MSD IS 0 IF
;ODD)
;
ALPHA   EQU     30
BETA    EQU     40
COUNT   EQU     5
        ORG     100H
        MOV     R0,#ALPHA
        MOV     R1,#BETA
        MOV     R2,#COUNT
        CLR     C
LP:     MOV     A,@R0
        ADDC    A,@R1
        DA      A
        MOV     @R0,A
        INC     R0
        INC     R1
        DJNZ    R2,LP
        END
```

The assembler may be run in two or three passes depending on available hardware. If the same device is used as both the list and punch device, three passes are necessary. Pass 1 builds the symbol table. Pass 2 produces the assembly listing and pass 3 produces the object code tape. Pass 4 combines passes 2 and 3 to produce both the listing and object file. Pass 1 must be run first; other passes may be run in any order and run more than once to produce multiple listing or object files. In this example, we show passes 1, 2, and 3.

```
P=1 XREF
```

Start pass 1 to build symbol table with XREF control selected. The cross-reference intermediate file is punched. This file must be input to the cross-reference-generator utility if a cross-reference listing is desired.

```
PASS 1 COMPLETE
P=2
```

Rewind source tape and start pass 2. The assembly listing is shown below. For a detailed explanation of each item in the listing, see Appendix D.

```
INTELLEC MONITOR MCS-48/UPI-41 ASSEMBLER, V1.0                 PAGE 1

LOC     OBJ     SEQ                 SOURCE STATEMENT

                  1     ;DECIMAL ADDITION ROUTINE. ADD BCD NUMBER
                  2     ;AT LOCATION 'BETA' TO BCD NUMBER AT 'ALPHA' WITH
                  3     ;RESULT IN 'ALPHA.' LENGTH OF NUMBER IS 'COUNT' DIGIT
                  4     ;PAIRS. (ASSUME BOTH BETA AND ALPHA ARE SAME LENGTH
                  5     ;AND HAVE EVEN NUMBER OF DIGITS OR MSD IS 0 IF
                  6     ;ODD)
                  7     ;
001E              8     ALPHA       EQU      30
0028              9     BETA        EQU      40
0005             10     COUNT       EQU      5
0100             11                 ORG      100H
0100    B81E     12                 MOV      R0,#ALPHA
0102    B928     13                 MOV      R1,#BETA
0104    BA05     14                 MOV      R2,#COUNT
0106    97       15                 CLR      C
0107    F0       16     LP:         MOV      A,@R0
0108    71       17                 ADDC     A,@R1
0109    57       18                 DA       A
010A    A0       19                 MOV      @R0,A
010B    18       20                 INC      R0
010C    19       21                 INC      R1
010D    EA07     22                 DJNZ     R2,LP
                 23                 END

USER SYMBOLS

ALPHA   001E    BETA    0028    COUNT   0005    LP      0107

ASSEMBLY COMPLETE, NO ERRORS

PASS 2 COMPLETE, NO ERRORS
P=3 NODEBUG
```

Rewind source tape and start pass 3 with object symbol table output suppressed. The actual object code is shown below in hexadecimal format. The assembler begins the object file by punching 120 null characters to provide 12 inches of leader and ends it with another 12 inches of blank trailer.

```
:0F010000B81EB928BA0597F07157A01819EA0769
:00000001FF

PASS 3 COMPLETE. NO ERRORS
P=E
```

With assembly completed, an exit command causes the assembler to transfer control to the Intellec monitor.

```
        (monitor prompt)
```

To obtain a cross-reference listing, load the paper tape reader with the cross-reference-generator.

```
.R0
```

Read the cross-reference generator into Intellec memory and load the paper tape output produced during pass 1.

```
.G20
```

Start execution by transferring control to hexadecimal address 20, the cross-reference generator's start address.

```
INTELLEC MONITOR ASSEMBLER SYMBOL CROSS REFERENCE, V1.0
```

The above sign-on message is sent to the console output device. By striking the carriage return, the paper tape is read and the following cross-reference listing is generated.

```
INTELLEC MONITOR ASSEMBLER SYMBOL CROSS REFERENCE, V1.0          PAGE 1

ALPHA         8#       12
BETA          9#       13
COUNT        10#       14
LP           16#       22

CROSS REFERENCE COMPLETE.
```

If an error is encountered during this listing, the listing is stopped immediately.

APPENDIXES

A. MCS-48 and UPI-41 Instruction Summary

B. Assembler Directive Summary

C. Assembler Control Summary

D. List File Formats

E. Reference Tables

F. Error Messages

A. MCS-48 AND UPI-41 INSTRUCTION SUMMARY

This appendix summarizes the MCS-48 and UPI-41 instruction sets. The instructions are first listed alphabetically by opcode (including binary encoding, number of cycles, system limitations, and description of function). They are then listed in order of hexadecimal opcode encoding.

SPECIAL OPERATORS AND RESERVED WORDS

The following special operators can be included in expressions in MCS-48 and UPI-41 instructions:

Operator	*Meaning*
+	Unary or binary addition.
–	Unary or binary subtraction.
*	Multiplication.
/	Division. Any remainder is discarded (7/3=2).
MOD	Modulo. Result is remainder produced by division operation (7 MOD 3 = 1).
SHR x	Logical shift right 'x' bit positions. No wraparound, zero fill.
SHL x	Logical shift left 'x' bit positions. No wraparound, zero fill.
NOT	Logical one's complement.
AND	Logical AND (=1 if both ANDed bits are 1).
OR	Logical OR (=1 if either ORed bit is 1).
XOR	Logical EXCLUSIVE OR (=1 if bits are different).
EQ	Logical equality.
NE	Logical inequality.
NUL	Logical null (ISIS-II assembler only).
LT	'Less than' relational operator.
LE	'Less than or equal' relational operator.
GT	'Greater than' relational operator.
GE	'Greater than or equal' relational operator.
HIGH	Isolate high-order 8 bits of a 16-bit value.
LOW	Isolate low-order 8 bits of a 16-bit value.

The '$' symbol, and the following opcodes, operands, and directives cannot be specified as user-defined symbols except in a local context.

Opcodes:

ADD
ADDC
ANL
ANLD
CALL
CLR
CPL
DA
DEC
DIS
DJNZ
EN
ENTO
IN
INC
INS
JBn
JC
JF0
JF1
JMP
JMPP
JNC
JNI
JNIBF
JNT0
JNT1
JNZ
JOBF
JTF
JT0
JT1
JZ
MOV
MOVD
MOVP
MOVP3
MOVX
NOP
ORL
ORLD
OUT
OUTL
RET
RETR
RL
RLC
RR
RRC
SEL
STOP
STRT
SWAP
XCH
XCHD
XRL

Operands:

A
AN0
AN1
BUS
C
CLK
CNT
DBB
F0
F1
FLAGS
I
MB0
MB1
P0
P1
P2
P4
P5
P6
P7
PSW
R0
R1
R2
R3
R4
R5
F6
R7
RAD
RB0
RB1
STS
T
TCNT
TCNTI

Directives:

DB
DS
DW
ELSE
END
ENDIF
ENDM
EOT
EQU
EXITM
IF
IRP
IRPC
LOCAL
MACRO
ORG
REPT
SET

MCS-48 AND UPI-41 ASSEMBLY LANGUAGE NOTATION

The following symbols and abbreviations are used to describe the functioning of MCS-48 and UPI-41 instructions.

A	accumulator
AC	auxiliary carry
addr	12-bit ROM/EPROM address
Bb	bit identifier (b=0-7)
BS	bank switch
BUS	BUS port
C	Carry
CLK	clock
CNT	event counter
D	4-bit digit
data	8-bit number or expression
DBB	data bus buffer
DBF	designate memory bank flip-flop
F0,F1	flag 0, flag 1
I	interrupt
IBF	input buffer flag
OBF	output buffer flag
P	mnemonic for ‘in-page’ operation
PC	program counter
Pp	port designator (p=0-2 or 4-7)
PSW	program status word
Rr	register designator (r=0,1 or 0-7)
SP	stack pointer
T	timer
TF	timer flag

In the following tables ‘8048’ also refers to the ‘8748,’ ‘8049,’ ‘8039,’ and ‘8035’ microcomputers. ‘8041’ also refers to the ‘8741’ microcomputer.

SUMMARY BY MNEMONIC OPCODE

Mnemonic	Binary Code		8048	8041	8021	Function
ADD A,#data (A)←(A)+data	00000011 dddddddd	2	X	X	X	Add immediate data to A. C and AC are affected.
ADD A,Rr (A)←(A)+(Rr) r=0-7	01101rrr	1	X	X	X	Add register data to A. C and AC are affected.
ADD A,@Rr (A)←(A)+((Rr)) r=0-1	0110000r	1	X	X	X	Add data in resident RAM location addressed by 'Rr' to A. C and AC are affected.
ADDC A,#data (A)←(A)+data+(C)	00010011 dddddddd	2	X	X	X	Add C and immediate data to A. C and AC are affected.
ADDC A,Rr (A)←(A)+(Rr)+(C) r=0-7	01111rrr	1	X	X	X	Add C and immediate data to A. C and AC are affected.
ADDC A,@Rr (A)←(A)+((Rr))+(C) r=0-1	0111000r	1	X	X	X	Add C and data in resident RAM location addressed by Rr to A. C and AC are affected
ANL A,#data (A)←(A) AND data	01010011 dddddddd	2	X	X	X	AND A data with immediate mask.
ANL A,Rr (A)←(A) AND (Rr) r=0-7	01011rrr	1	X	X	X	AND A data with mask in Rr.
ANL A,@Rr (A)←(A) AND ((Rr)) r=0-1	0101000r	1	X	X	X	AND A data with mask in resident RAM location address by Rr.
ANL BUS,#data (BUS)←(BUS) AND data	10011000 dddddddd	2	X			AND BUS data with immediate mask.
ANL Pp,#data (Pp)←(Pp) AND data p=1-2	100110pp dddddddd	2	X	X		AND port p data with immediate mask.
ANLD Pp,A (Pp)←(Pp)AND(A0-3) p=4-7	100111pp	2	X	X	X	AND port p data with mask in A bits 0-3.

Mnemonic	Binary Code	Cycles	8048	8041	8021	Function
CALL addr ((SP))←(PC), (PSW4-7) (SP)←(SP)+1 (PC8-10)←addr 8-10 (PC0-7)←addr 0-7 (PC11)←DBF	10 8 aaa10100 7 0 aaaaaaaa	2	X	X	X	Store PC and PSW bits 4-7 in stack. Increment stack pointer. Transfer control to subroutine at location addr. PC 10-11 must be zero for 8041 and 8021
CLR A A←0	00100111	1	X	X	X	Clear A to zero.
CLR C C←0	10010111	1	X	X	X	Clear C to zero.
CLR F0 (F0)←0	10000101	1	X	X		Clear F0 to zero.
CLR F1 (F1)←0	10100101	1	X	X		Clear F1 to zero.
CPL A (A)←NOT (A)	00110111	1	X	X	X	One's complement A contents.
CPL C (C)←NOT (C)	10100111	1	X	X	X	Complement C.
CPL F0 (F0)←NOT (F0)	10010101	1	X	X		Complement F0.
CPL F1 (F1)←NOT (F1)	10110101	1	X	X		Complement F1.
DA A	01010111	1	X	X	X	A contents adjusted to form 2 BCD digits. C is affected.
DEC A (A)←(A)−1	00000111	1	X	X	X	Decrement A by 1.
DEC Rr (Rr)←(Rr)−1 r=0-7	11001rrr	1	X	X		Decrement Rr by 1.
DIS I	00010101	1	X	X		Disable external interrupt (8048). Disable write interrupt (8041).
DIS TCNTI	00110101	1	X	X		Disable timer/counter interrupt.

Mnemonic	Binary Code	Cycles	8048	8041	8021	Function
DJNZ Rr, addr (Rr)←(Rr)−1 r=0-7 If Rr NOT 0, (PC 0-7)←addr	11101rrr aaaaaaaa	2	X	X	X	Decrement Rr by 1. If Rr NOT 0, jump to addr.
EN I	00000101	1	X	X		Enable external interrupt (8048). Enable write interrupt (8041).
EN TCNTI	00100101	1	X	X		Enable timer/counter interrupt.
ENT0 CLK	01110101	1	X			Enable T0 as internal oscillator output.
IN A,DBB (A)←(DBB)	00100010	1		X		Input DBB data to A. Clear IBF.
IN A,Pp (A)←(Pp) p=0-2	000010pp	2	X	X	X	Input port p data to A. P0 used for 8021 only.
INC A (A)←(A)+1	00010111	1	X	X	X	Increment A by 1.
INC Rr (Rr)←(Rr)+1 r=0-7	00011rrr	1	X	X	X	Increment Rr by 1.
INC @Rr ((Rr))←((Rr))+1 r=0-1	0001000r	1	X	X	X	Increment resident RAM location addressed by Rr by 1.
INS A,BUS (A)←(BUS)	00001000	2	X			Read BUS with RD strobe and input contents to A.
JBb addr b=0-7 If Bb=1 (PC 0-7)←addr	bbb10010 aaaaaaaa	2	X	X		Jump to addr if bit b of A is 1.
JC addr If C=1, (PC 0-7)←addr	11110110 aaaaaaaa	2	X	X	X	Jump to addr if C=1.
JF0 addr If F0=1, (PC 0-7)←addr	10110110 aaaaaaaa	2	X	X		Jump to addr if F0=1

Mnemonic	Binary Code	Cycles	8048	8041	8021	Function
JF1 addr If F1=1, (PC 0-7)←addr	01110110	2	X	X		Jump to addr if F1=1.
JMP addr (PC8-10)←addr 8-10 (PC0-7)←addr 0-7 (PC11)←(DBF)	10 8 aaa 00100 7 0 aaaaaaaa	2	X	X	X	Jump to addr unconditionally. PC 10-11 must be zero for 8041 and 8021.
JMPP @A (PC 0-7)←((A))	10110011	2	X	X	X	The contents of the program memory location pointed to by A are substituted for PC bits 0-7.
JNC addr If C=0, (PC 0-7)←addr	11100110 aaaaaaaa	2	X	X	X	Jump to addr if C=0.
JNI addr If I=0, (PC 0-7)← addr	10000110 aaaaaaaa	2	X			Jump to addr if interrupt input goes low (I=0).
JNIBF addr If IBF=0, (PC 0-7)←addr	11010110 aaaaaaaa	2		X		Jump to addr if IBF=0.
JNT0 addr If T0=0, (PC 0-7)←addr	00100110 aaaaaaaa	2	X	X		Jump to addr if T0=0.
JNT1 addr If T1=0, (PC 0-7)←addr	01000110 aaaaaaaa	2	X	X	X	Jump to addr if T1=0.
JNZ addr If A≠0, (PC 0-7)←addr	10010110 aaaaaaaa	2	X	X	X	Jump to addr if A contents are not zero.
JOBF addr If OBF=1, (PC 0-7)←addr	10000110 aaaaaaaa	2		X		Jump to addr if OBF=1.
JTF addr If TF=1, (PC 0-7)←addr	00010110 aaaaaaaa	2	X	X	X	Jump to addr if TF=1.

Mnemonic	Binary Code	Cycles	8048	8041	8021	Function
JT0 addr If T0=1, (PC 0-7)←addr	00110110 aaaaaaaa	2	X	X		Jump to addr if T0=1.
JT1 addr If T1=1, (PC 0-7)←addr	01010110 aaaaaaaa	2	X	X	X	Jump to addr if T1=1.
JZ addr If A=0, (PC 0-7)←addr	11000110 aaaaaaaa	2	X	X	X	Jump to addr if A contents are zero.
MOV A,#data (A)←data	00100011 dddddddd	2	X	X	X	Move immediate data into A.
MOV A,PSW (A)←(PSW)	11000111	1	X	X		Move PSW data into A.
MOV A,Rr (A)←(Rr) r=0-7	11111rrr	1	X	X	X	Move data in Rr into A.
MOV A,@Rr (A)←((Rr)) r=0-1	1111000r	1	X	X	X	Move data in resident RAM location addressed by Rr into A.
MOV A,T (A)←(T)	01000010	1	X	X	X	Move data in timer into A.
MOV PSW,A (PSW)←(A)	11010111	1	X	X		Move data in A into PSW.
MOV Rr,A (Rr)←(A) r=0-7	10101rrr	1	X	X	X	Move data in A into Rr.
MOV Rr,#data (Rr)←data r=0-7	10111rrr dddddddd	2	X	X	X	Move immediate data into Rr.
MOV @Rr,A ((Rr))←(A) r=0-1	1010000r	1	X	X	X	Move data in A into resident RAM location addressed by Rr.
MOV @Rr,#data ((Rr))←data r=0-1	1011000r dddddddd	2	X	X	X	Move immediate data into resident RAM location addressed by Rr.

Mnemonic	Binary Code	Cycles	8048	8041	8021	Function
MOV T,A (T)←(A)	01100010	1	X	X	X	Move data in A into timer.
MOVD A,Pp (A 0-3)←Pp (A 4-7)←0 p=4-7	000011pp	2	X	X	X	Move data in 8243 port p into A bits 0-3. Zero A bits 4-7.
MOVD Pp,A (Pp)←(A 0-3) p=4-7	001111pp	2	X	X	X	Move data in A into 8243 port p.
MOVP A,@A (PC 0-7)←(A) (A)←((PC))	10100011	2	X	X	X	Move data in program memory location addressed by A into A. Program counter is restored.
MOVP3 A,@A (PC0-7)←(A) (PC8-10)←011B (A)←((PC))	11100011	2	X	X		Move data in program memory page 3 location addressed by A into A. Program counter is restored.
MOVX A,@Rr (A)←((Rr)) r=0-1	1000000r	2	X			Move data in external RAM location addressed by Rr into A.
MOVX @Rr,A ((Rr))←A r=0-1	1001000r	2	X			Move data in A into external RAM location addressed by Rr.
NOP	00000000	1	X	X	X	No operation.
ORL A,#data (A)←(A) OR data	01000011 dddddddd	2	X	X	X	OR contents of A with data mask.
ORL A,Rr (A)←(A) OR (Rr) r=0-7	01001rrr	1	X	X	X	OR data in A with Rr mask.
ORL A,@Rr (A)←(A) OR ((Rr)) r=0-1	0100000r	1	X	X	X	OR data in A with mask in resident RAM location addressed by Rr.
ORL BUS,#data (BUS)←(BUS) OR data	10001000 dddddddd	2	X			OR contents of BUS with data mask.

Mnemonic	Binary Code	Cycles	8048	8041	8021	Function
ORL Pp,#data (Pp)←(Pp) OR data p=1-2	100010pp dddddddd	2	X	X		OR contents of port p with data mask.
ORLD Pp,A (Pp)←(Pp) OR (A0-3) p=4-7	100011pp	2	X	X	X	OR data in 8243 port p with mask in A bits 0-3.
OUT DBB,A (DBB)←(A)	00000010	1		X		Output data in A to DBB. Set OBF.
OUTL BUS,A (BUS)←(A)	00000010	2	X			Output data in A to BUS and latch.
OUTL P0,A (P0) ← (A)	10010000	2			X	Output data in A to port 0 and latch.
OUTL Pp,A (Pp)←(A) p=1-2	001110pp	2	X	X		Output data in A to port p and latch.
RET (SP)←(SP)−1 (PC)←((SP))	10000011	2	X	X	X	Restore program counter from stack and return to main routine.
RETR (SP)←(SP)−1 (PC)←((SP)) (PSW4-7)←((SP))	10010011	2	X	X		Restore program counter and PSW bits 4-7 from stack and return to main routine. Reenable interrupts if the interrupt enable flip-flop if set.
RL A (An+1)←(An) (A0)←(A7) n=0-6	11100111	1	X	X	X	Rotate A left. C is unaffected.
RLC A (An+1)←(An) (A0)←(C) (C)←(A7) n=0-6	11110111	1	X	X	X	Rotate A left, through C.
RR A (An)←(An+1) (A7)←(A0) n=0-6	01110111	1	X	X	X	Rotate A right. C is unaffected.
RRC A (An)←(An+1) (A7)←(C) (C)←(A0) n=0-6	011001111	1	X	X	X	Rotate A right, through C.

Mnemonic	Binary Code	Cycles	8048	8041	8021	Function
SEL MB0 (DBF)←0	11100101	1	X			Select program memory bank 0.
SEL MB1 (DBF)←1	11110101	1	X			Select program memory bank 1.
SEL RB0 (BS)←0	11000101	1	X	X		Select working register bank 0.
SEL RB1 (BS)←1	11010101	1	X	X		Select working register bank 1.
STOP TCNT	01100101	1	X	X	X	Stop timer or disable event counter.
STRT CNT	01000101	1	X	X	X	Enable T1 as event counter input and start.
STRT T	01010101	1	X	X	X	Clear timer prescaler and start timer.
SWAP A (A4-7)↔(A0-3)	01000111	1	X	X	X	Swap A bits 0-3 with A bits 4-7.
XCH A,Rr (A)↔(Rr) r=0-7	00101 r r r	1	X	X	X	Exchange contents of A and Rr.
XCH A,@Rr (A)↔((Rr)) r=0-1	0010000 r	1	X	X	X	Exchange contents of A and resident RAM location addressed by Rr.
XCHD A,@Rr (A0-3)↔((Rr0-3)) r=0-1	0011000 r	1	X	X	X	Exchange A bits 0-3 with bits 0-3 of resident RAM location addressed by Rr.
XRL A,#data (A)←(A) XOR data	11010011 dddddddd	2	X	X	X	XOR contents of A with data mask.
XRL A,Rr (A)←(A) XOR (Rr) r=0-7	11011 r r r	1	X	X	X	XOR data in A with mask in Rr.
XRL A,@Rr (A)←(A) XOR ((Rr)) r=0-1	1101000 r	1	X	X	X	XOR data in A with mask in resident RAM location addressed by Rr.

SUMMARY BY HEXADECIMAL OPCODE

Code	Mnemonic	8048	8041	8021	Code	Mnemonic	8048	8041	8021
00	NOP	X	X	X	20	XCH A,@R0	X	X	X
01	undefined	–	–	–	21	XCH A,@R1	X	X	X
02	OUTL BUS,A	X	–	–	22	IN A,DBB	–	X	–
02	OUT DBB,A	–	X	–					
03	ADD A,#data	X	X	X	23	MOV A,#data	X	X	X
04	JMP (Page 0)	X	X	X	24	JMP (Page 1)	X	X	X
05	EN I	X	X	–	25	EN TCNTI	X	X	–
06	undefined	–	–	–	26	JNT0	X	X	–
07	DEC A	X	X	X	27	CLR A	X	X	X
08	INS A, BUS	X	–	–	28	XCH A,R0	X	X	X
08	IN A,P0	–	–	X					
09	IN A,P1	X	X	X	29	XCH A,R1	X	X	X
0A	IN A,P2	X	X	X	2A	XCH A,R2	X	X	X
0B	undefined	–	–	–	2B	XCH A,R3	X	X	X
0C	MOVD A,P4	X	X	X	2C	XCH A,R4	X	X	X
0D	MOVD A,P5	X	X	X	2D	XCH A,R5	X	X	X
0E	MOVD A,P6	X	X	X	2E	XCH A,R6	X	X	X
0F	MOVD A,P7	X	X	X	2F	XCH A,R7	X	X	X
10	INC @R0	X	X	X	30	XCHD A,@R0	X	X	X
11	INC @R1	X	X	X	31	XCHD A,@R1	X	X	X
12	JB0	X	X	–	32	JB1	X	X	–
13	ADDC A,#data	X	X	X	33	undefined	–	–	–
14	CALL (Page 0)	X	X	X	34	CALL (Page 1)	X	X	X
15	DIS I	X	X	–	35	DIS TCNTI	X	X	–
16	JTF	X	X	X	36	JT0	X	X	–
17	INC A	X	X	X	37	CPL A	X	X	X
18	INC R0	X	X	X	38	undefined	–	–	–
19	INC R1	X	X	X	39	OUTL P1,A	X	X	X
1A	INC R2	X	X	X	3A	OUTL P2,A	X	X	X
1B	INC R3	X	X	X	3B	undefined	–	–	–
1C	INC R4	X	X	X	3C	MOVD P4,A	X	X	X
1D	INC R5	X	X	X	3D	MOVD P5,A	X	X	X
1E	INC R6	X	X	X	3E	MOVD P6,A	X	X	X
1F	INC R7	X	X	X	3F	MOVD P7,A	X	X	X

Code	Mnemonic	8048	8041	8021	Code	Mnemonic	8048	8041	8021
40	ORL A,@R0	X	X	X	60	ADD A,@R0	X	X	X
41	ORL A,@R1	X	X	X	61	ADD A,@R1	X	X	X
42	MOV A,T	X	X	X	62	MOV T,A	X	X	X
43	ORL A,#data	X	X	X	63	undefined	–	–	–
44	JMP (Page 2)	X	X	X	64	JMP (Page 3)	X	X	X
45	STRT CNT	X	X	X	65	STOP TCNT	X	X	X
46	JNT1	X	X	X	66	undefined	–	–	–
47	SWAP A	X	X	X	67	RRC A	X	X	X
48	ORL A,R0	X	X	X	68	ADD A,R0	X	X	X
49	ORL A,R1	X	X	X	69	ADD A,R1	X	X	X
4A	ORL A,R2	X	X	X	6A	ADD A,R2	X	X	X
4B	ORL A,R3	X	X	X	6B	ADD A,R3	X	X	X
4C	ORL A,R4	X	X	X	6C	ADD A,R4	X	X	X
4D	ORL A,R5	X	X	X	6D	ADD A,R5	X	X	X
4E	ORL A,R6	X	X	X	6E	ADD A,R6	X	X	X
4F	ORL A,R7	X	X	X	6F	ADD A,R7	X	X	X
50	ANL A,@R0	X	X	X	70	ADDC A,@R0	X	X	X
51	ANL A,@R1	X	X	X	71	ADDC A,@R1	X	X	X
52	JB2	X	X	–	72	JB3	X	X	–
53	ANL A,#data	X	X	X	73	undefined	–	–	–
54	CALL (Page 2)	X	X	X	74	CALL (Page 3)	X	X	X
55	STRT T	X	X	X	75	ENT0 CLK	X	–	–
56	JT1	X	X	X	76	JF1	X	X	–
57	DA A	X	X	X	77	RR A	X	X	X
58	ANL A,R0	X	X	X	78	ADDC A,R0	X	X	X
59	ANL A,R1	X	X	X	79	ADDC A,R1	X	X	X
5A	ANL A,R2	X	X	X	7A	ADDC A,R2	X	X	X
5B	ANL A,R3	X	X	X	7B	ADDC A,R3	X	X	X
5C	ANL A,R4	X	X	X	7C	ADDC A,R4	X	X	X
5D	ANL A,R5	X	X	X	7D	ADDC A,R5	X	X	X
5E	ANL A,R6	X	X	X	7E	ADDC A,R6	X	X	X
5F	ANL A,R7	X	X	X	7F	ADDC A,R7	X	X	X

Code	Mnemonic	8048	8041	8021	Code	Mnemonic	8048	8041	8021
80	MOVX A,@R0	X	–	–	A0	MOV @R0,A	X	X	X
81	MOVX A,@R1	X	–	–	A1	MOV @R1,A	X	X	X
82	undefined	–	–	–	A2	undefined	–	–	–
83	RET	X	X	X	A3	MOVP A,@A	X	X	X
84	JMP (Page 4)	X	–	–	A4	JMP (Page 5)	X	–	–
85	CLR F0	X	X	–	A5	CLR F1	X	X	–
86	JNI	X	–	–	A6	undefined	–	–	–
86	JOBF	–	X	–					
87	undefined	–	–	–	A7	CPL C	X	X	X
88	ORL BUS,#data	X	–	–	A8	MOV R0,A	X	X	X
89	ORL P1,#data	X	X	–	A9	MOV R1,A	X	X	X
8A	ORL P2,#data	X	X	–	AA	MOV R2,A	X	X	X
8B	undefined	–	–	–	AB	MOV R3,A	X	X	X
8C	ORLD P4,A	X	X	X	AC	MOV R4,A	X	X	X
8D	ORLD P5,A	X	X	X	AD	MOV R5,A	X	X	X
8E	ORLD P6,A	X	X	X	AE	MOV R6,A	X	X	X
8F	ORLD P7,A	X	X	X	AF	MOV R7,A	X	X	X
90	MOVX @R0,A	X	–	–	B0	MOV @R0,#data	X	X	X
90	OUTL P0,A	–	–	X					
91	MOVX @R1,A	X	–	–	B1	MOV @R1,#data	X	X	X
92	JB4	X	X	–	B2	JB5	X	X	–
93	RETR	X	X	–	B3	JMPP @A	X	X	X
94	CALL (Page 4)	X	–	–	B4	CALL (Page 5)	X	–	–
95	CPL F0	X	X	–	B5	CPL F1	X	X	–
96	JNZ	X	X	X	B6	JF0	X	X	–
97	CLR C	X	X	X	B7	undefined	–	–	–
98	ANL BUS,#data	X	–	–	B8	MOV R0,#data	X	X	X
99	ANL P1,#data	X	X	–	B9	MOV R1,#data	X	X	X
9A	ANL P2,#data	X	X	–	BA	MOV R2,#data	X	X	X
9B	undefined	–	–	–	BB	MOV R3,#data	X	X	X
9C	ANLD P4,A	X	X	X	BC	MOV R4,#data	X	X	X
9D	ANLD P5,A	X	X	X	BD	MOV R5,#data	X	X	X
9E	ANLD P6,A	X	X	X	BE	MOV R6,#data	X	X	X
9F	ANLD P7,A	X	X	X	BF	MOV R7,#data	X	X	X

Code	Mnemonic	8048	8041	8021	Code	Mnemonic	8048	8041	8021
C0	undefined	–	–	–	E0	undefined	–	–	–
C1	undefined	–	–	–	E1	undefined	–	–	–
C2	undefined	–	–	–	E2	undefined	–	–	–
C3	undefined	–	–	–	E3	MOVP3 A,@A	X	X	–
C4	JMP (Page 6)	X	–	–	E4	JMP (Page 7)	X	–	–
C5	SEL RB0	X	X	–	E5	SEL MB0	X	–	–
C6	JZ	X	X	X	E6	JNC	X	X	X
C7	MOV A,PSW	X	X	–	E7	RL A	X	X	X
C8	DEC R0	X	X	–	E8	DJNZ R0,addr	X	X	X
C9	DEC R1	X	X	–	E9	DJNZ R1,addr	X	X	X
CA	DEC R2	X	X	–	EA	DJNZ R2,addr	X	X	X
CB	DEC R3	X	X	–	EB	DJNZ R3,addr	X	X	X
CC	DEC R4	X	X	–	EC	DJNZ R4,addr	X	X	X
CD	DEC R5	X	X	–	ED	DJNZ R5,addr	X	X	X
CE	DEC R6	X	X	–	EE	DJNZ R6,addr	X	X	X
CF	DEC R7	X	X	–	EF	DJNZ R7,addr	X	X	X
D0	XRL A,@R0	X	X	X	F0	MOV A,@R0	X	X	X
D1	XRL A,@R1	X	X	X	F1	MOV A,@R1	X	X	X
D2	JB6	X	X	–	F2	JB7	X	X	–
D3	XRL A,#data	X	X	X	F3	undefined	–	–	–
D4	CALL (Page 6)	X	–	–	F4	CALL (Page 7)	X	–	–
D5	SEL RB1	X	X	–	F5	SEL MB1	X	–	–
D6	JNIBF	–	X	–	F6	JC	X	X	X
D7	MOV PSW, A	X	X	–	F7	RLC A	X	X	X
D8	XRL A,R0	X	X	X	F8	MOV A,R0	X	X	X
D9	XRL A,R1	X	X	X	F9	MOV A,R1	X	X	X
DA	XRL A,R2	X	X	X	FA	MOV A,R2	X	X	X
DB	XRL A,R3	X	X	X	FB	MOV A,R3	X	X	X
DC	XRL A,R4	X	X	X	FC	MOV A,R4	X	X	X
DD	XRL A,R5	X	X	X	FD	MOV A,R5	X	X	X
DE	XRL A,R6	X	X	X	FE	MOV A,R6	X	X	X
DF	XRL A,R7	X	X	X	FF	MOV A,R7	X	X	X

B. ASSEMBLER DIRECTIVE SUMMARY

Assembler directives are summarized alphabetically in this appendix. The following terms are used to describe the contents of directive fields.

NOTATION

Term	*Interpretation*
Expression	Numerical expression evaluated during assembly; must evaluate to 8 or 16 bits depending on directive issued.
List	Series of symbolic values or expressions, separated by commas.
Name	Symbol name terminated by a space.
Null	Field must be empty or an error results.
Oplab	Optional label; must be terminated by a colon.
Parameter	Dummy parameters are symbols holding the place of actual parameters (symbolic values or expressions) specified elsewhere in the program.
String	Series of ASCII characters, surrounded by single quote marks.
Text	Series of ASCII characters.

Macro definitions and calls allow the use of the special characters listed below.

Character	*Function*
&	Ampersand. Used to concatenate symbols.
⟨ ⟩	Angle brackets. Used to delimit text, such as lists, that contain other delimiters.
;;	Double semicolon. Used before a comment in a macro definition to prevent inclusion of the comment in each macro expansion.

Character	*Function*
!	Exclamation point (escape character). Placed before a delimiter to be passed as a literal in an actual parameter. To pass a literal exclamation point, issue ‘!!.’
%	Percent sign. Precedes actual parameters to be evaluated immediately when the macro is called.

SUMMARY OF DIRECTIVES

FORMAT			FUNCTION
Label	Opcode	Operand(s)	
oplab:	DB	exp(s) or string(s)	Define 8-bit data byte(s). Expressions must evaluate to one byte.
oplab:	DS	expression	Reserve data storage area of specified length.
oplab:	DW	exp(s) or string(s)	Define 16-bit data word(s). Strings limited to 1-2 characters.
oplab:	ELSE	null	Conditional assembly. Code between ELSE and ENDIF directives is assembled if expression in IF clause is FALSE. (See IF.)
oplab:	END	expression	Terminate assembler pass. Must be last statement of program. Program execution starts at ‘exp,’ if present; otherwise, at location 0.
oplab:	ENDIF	null	Terminate conditional assembly block.
oplab:	EOT	null	Specify end of paper tape.
name	EQU	expression	Define symbol ‘name’ with value ‘exp.’ Symbol is not redefinable.
oplab:	IF	expression	Assemble code between IF and following ELSE or ENDIF directive if ‘exp’ is true.
oplab:	ORG	expression	Set location counter to ‘expression.’
name	SET	expression	Define symbol ‘name’ with value ‘expression.’ Symbol can be redefined.

MACRO DIRECTIVES

FORMAT			FUNCTION
Label	Opcode	Operand(s)	
null	ENDM	null	Terminate macro definition.
oplab:	EXITM	null	Alternate terminator of macro definition. (See ENDM.)
oplab:	IRP	dummy param,<list>	Repeat instruction sequence, substituting one element from 'list' for 'dummy param' in each iteration.
oplab:	IRPC	dummy param, text	Repeat instruction sequence, substituting one character from 'text' for 'dummy param' in each iteration.
null:	LOCAL	label name(s)	Specify label(s) in macro definition to have local scope.
name	MACRO	dummy param(s)	Define macro 'name' and dummy parameter(s) to be used in macro definition.
oplab:	REPT	expression	Repeat REPT block 'expression' times.

C. ASSEMBLER CONTROL SUMMARY

CONTROL FORMATS

Assembler controls can be specified during the assembly operation, or can be embedded as control lines in the source file.

The possible formats are:

[:Fn:]ASM48 sourcefile control-list	(ISIS-II assembly-time control specification)
P=n control-list	(MONITOR assembly-time control specification)
$control-list	(control line)

Items in the control list are separated by blanks. For control lines, the '$' preceding the first control must be in column 1. All controls must be spelled out in their entirety. Primary controls must be specified before the first source file statement.

COMMON CONTROLS

The following controls are common to both the ISIS-II and MONITOR assemblers.

Control	*Prim/Gen*	*Description*
COND	General	Include conditionally-skipped source code and directives in assembly listing.
DEBUG	Primary	Include symbol table in object file.
EJECT	General	Skip to logical top-of-form.
LIST	General	ISIS-II: Generate assembly listing to file specified by PRINT. MONITOR: Enable printout of assembly listing during pass 2 or pass 4.
MOD21	Primary	Recognize 8021 instruction set.
MOD41	Primary	Recognize UPI-41 instruction set.
NOCOND	General	Suppress listing of conditionally-skipped source code.

Control	*Prim/Gen*	*Description*
NODEBUG	Primary	Suppress inclusion of symbol table in object file.
NOLIST	General	ISIS-II: Suppress listing of PRINT file, except header, symbol and XREF tables and errors. MONITOR: Negate effect of pass 2 or pass 4 listing request, except header, symbol and XREF tables, and errors.
NOPAGING	Primary	Assembly listing is not broken into separate pages.
NOSYMBOLS	Primary	Suppress listing of symbol table.
NOXREF	Primary	Suppress generation of symbol-cross-reference file.
PAGELENGTH(n)	Primary	Set page length to 'n' lines.
PAGEWIDTH(n)	Primary	Set page width to 'n' characters.
PAGING	Primary	Break assembly listing into pages; repeat headers each page break.
SYMBOLS	Primary	Include symbol table in assembly list.
TITLE('str')	General	Print string (up to 64 characters) as second line of page header on assembly listing.
XREF	Primary	Create symbol-cross-reference file.

ISIS-II ASSEMBLER CONTROLS

The following controls are unique to the ISIS-II assembler.

Control	*Prim/Gen*	*Description*
GEN	General	Include macro expansion source text in assembly listing.
INCLUDE(file)	General	Inculde specified source file in file being processed.
MACRODEBUG	General	Include assembler-generated symbols in assembly listing and object file.
MACROFILE(dr)	Primary	Run program containing macros; direct macro temporary files to specified drive
NOGEN	General	Suppress listing of macro expansion source text.
NOMACRODEBUG	General	Suppress output of assembler-generated macro symbols.
NOMACROFILE	Primary	Do not create macro temporary files.
NOOBJECT	Primary	Suppress generation of object file.
NOPRINT	Primary	Suppress generation of assembly listing file.
OBJECT(file)	Primary	Create object file on specified device.
PRINT(file)	Primary	Create assembly listing file on specified device.
RESTORE	General	Restore most recent command set from command stack.
SAVE	General	Save current setting of LIST, COND, and GEN controls in command stack.

MONITOR ASSEMBLER PASS OPTIONS

The following are valid responses to the MONITOR assembler's 'pass number' prompt (P=). The first pass number specified must be '1.'

Pass	*Description*
1	Assembler builds symbol table.
2	Assembly listing is generated.
3	Object file is punched into paper tape.
4	Object file and assembly listing are generated. (Must be on separate devices.)
E	Exit to Intellec monitor.

DEFAULTS

ISIS-II Assembler	*MONITOR Assembler*
COND	COND
GEN	LIST
LIST	NODEBUG
NODEBUG	NOXREF
NOMACRODEBUG	PAGELENGTH(66)
NOMACROFILE	PAGEWIDTH(120)
NOXREF	PAGING
OBJECT(source.HEX)	SYMBOLS
PAGELENGTH(66)	
PAGEWIDTH(120)	
PAGING	
PRINT(source.LST)	
SYMBOLS	

D. LIST FILE FORMATS

ASSEMBLY LISTING FORMAT

The assembly listing format is essentially the same for both the ISIS-II and Intellec MONITOR versions of the assembler. The list file is designed for output to a line printer or terminal. Unless otherwise specified, an output page consists of 66 lines, 120 characters wide, including three leading and three trailing blank lines, the page header, title line, column headings, and assembly output lines. If a listing line exceeds the right margin setting, it is continued in column 25 of the following line (unless the line exceeds 132 characters, in which case those)132 are truncated).

For the ISIS-II assembler only, the first line of the first page of a listing is an echo of the ISIS-II call to the assembler followed by the page header.

If the NOPAGING assembler control is selected, the page header is followed by the title line and column heading, and finally the complete assembly listing with no additional headers.

Page Header

Columns	*Description*
1-40	The string 'ISIS-II MCS-48/UPI-41 MACRO ASSEMBLER' or The string 'INTELLEC MONITOR MCS-48/UPI-41 ASSEMBLER'
41	Blank.
42-45	The string 'Vx.y' where 'x' is the version and 'y' is the release number.
46-64	Blanks.
65-68	The string 'PAGE '
69	Blank.
70-72	Three character positions containing the page number in decimal.

Title Line

Columns	*Description*
1-64	Program title as specified in TITLE assembler control

Column Heading

Columns	*Description*
1-2	Blanks.
3-5	The string 'LOC'
6-7	Blanks.
8-10	The string 'OBJ'
11-16	Blanks.
17-19	The string 'SEQ'
31-46	The string 'SOURCE STATEMENT'

Assembly Output Line

Columns	*Description*
1	Assembler error code. If the assembler encountered an error in this source line, the appropriate error code appears in this column. Otherwise, this column is blank. If an error occurs in the present line, the following line will be blank except for a decimal sequence number in columns 3-6 enclosed by parentheses. This sequence number is a pointer to the previous line containing an error. The first error encountered in a program will be followed by a line with a pointer equal to zero. See Appendix F for error codes.
2	Blank.
3-6	The address assigned to the first byte of the object code shown in columns 8-9 of this line is printed in hexadecimal. In addition, the result of the value-generating assembler directives ORG, EQU, SET, and END will appear in this field. For END, the program start address value will appear in this field if specified; otherwise blank.

Columns	*Description*
7	Blank.
8-9	The first byte of object code produced by the assembler for this source line is printed here in hexadecimal. If the source statement produces no object code (comments and assembler directives), this field is blank.
10-11	Second byte of object code in hexadecimal. This field will be blank if the source statement generates only one byte of object code or no object code.
12-13	Third byte of object code in hexadecimal, if generated.
14-15	Fourth byte of object code in hexadecimal, if generated.
16-17	Blanks.
18	(ISIS assembler only.) Blank if no nested source INCLUDE files; otherwise, the number 1-4 indicating the level of nesting.
19	(ISIS assembler only.) Blank if not listing a source INCLUDE file; otherwise an '=' sign.
20-23	Four character positions containing the source line number in decimal, right-justified and left blank-filled.
24	(ISIS assembler only.) Macro expansion flag. A '+' in this column indicates that the source line was produced as a result of a macro expansion. Otherwise, this column will be blank.
25- . . .	Listing of assembler source text. This field terminates at column 72 for most output devices other than the line printer. For a line printer, this field terminates at column 132.

For DB and DW assembler directives containing a *list* of operands, the generated code for each operand will be listed on a separate line.

If a list line exceeds the specified page width, the source line continues starting at column 25 of the next line.

Symbol Table Listing

The listing of the assembled source code is followed by an optional symbol table listing. If the NOSYMBOLS control is specified, the symbol table listing is suppressed.

The symbol table is preceded by the title 'USER SYMBOLS' in columns 1-12 of the listing. The format of a symbol table output line is as follows:

Columns	*Description*
1-6	Symbol name up to six characters, left justified.
7	Blank.
8-11	Symbol value in hexadecimal.
12-15	Blanks.
16-n	Repetition of columns 1-15 format where 'n' is the pagewidth.

Error Summary

After listing the last line of the symbol table and spacing one line, the assembler lists an error summary line in the following format:

Columns	*Description*
1-19	The string 'ASSEMBLY COMPLETE'
20-23	Number of errors. Four character positions containing the number of errors in the source encountered during assembly. This number is output in decimal, right-justified and left blank-filled. If there are no errors, the string 'NO' is output instead.
24	Blank.
25-30	The string 'ERRORS' (or 'ERROR' if only one error in program).
31	Blank.
32	If the number of errors is not zero, the character '('; otherwise, blank.
33-36	If the number of errors is not zero, these four character positions contain the sequence number (in decimal) of the last source line with an error; otherwise, blank.
37	If the number of errors is not zero, the character ')'; otherwise, blank.

SYMBOL-CROSS-REFERENCE LISTING

Both assemblers generate a file of symbol-cross-reference records during assembly pass 1 if the XREF assembler control is selected. This control is described in Chapter 8. The actual symbol-cross-reference listing is generated by running either the paper-tape-resident or disk-resident version of the XREF utility program, using the file created during pass 1 as input. The utility sorts symbols alphabetically before producing its listing.

Page Header

Columns	*Description*
1-49	The string 'ISIS-II ASSEMBLER SYMBOL CROSS REFERENCE' or The string 'INTELLEC MONITOR ASSEMBLER SYMBOL CROSS REFERENCE'
50	Blank.
51-54	The string 'Vx.y' where 'x' is the version and 'y' is the release number.
55-65	Blanks.
65-68	The string 'PAGE'
69	Blank.
70-72	Three character positions containing the page number in decimal.

Cross-Reference Output Line

Columns	*Description*
1-6	For lines that start a new entry, this field contains the symbol itself; otherwise, blank.
7	Blank.
8-11	Sequence number of source line containing a reference to or definition of the current symbol entry.
12	Blank if the source line contains a reference; '#' if the source line contains a definition.
13-14	Blanks.
15-68	Repetitions of format in columns 1-14.

If no errors are found during the symbol-cross-reference listing, the message

CROSS REFERENCE COMPLETE

is issued. If an error is found, the listing terminates immediately.

E. REFERENCE TABLES

This appendix contains the following general reference tables:

- ASCII codes
- Powers of two
- Powers of 16 (in base 10)
- Powers of 10 (in base 16)
- Hexadecimal-decimal integer conversion

ASCII CODES

The 8048 uses the 7-bit ASCII code, with the high-order 8th bit (parity bit) always reset.

GRAPHIC OR CONTROL	ASCII (HEXADECIMAL)
NUL	00
SOH	01
STX	02
ETX	03
EOT	04
ENQ	05
ACK	06
BEL	07
BS	08
HT	09
LF	0A
VT	0B
FF	0C
CR	0D
SO	0E
SI	0F
DLE	10
DC1 (X-ON)	11
DC2 (TAPE)	12
DC3 (X-OFF)	13
DC4 (~~TAPE~~)	14
NAK	15
SYN	16
ETB	17
CAN	18
EM	19
SUB	1A
ESC	1B
FS	1C
GS	1D
RS	1E
US	1F
SP	20
!	21
ʼʼ	22
#	23
$	24
%	25
&	26
/	27
(	28
)	29
*	2A
+	2B
,	2C
–	2D
.	2E
/	2F
0	30
1	31
2	32
3	33
4	34
5	35
6	36
7	37
8	38
9	39
:	3A
;	3B
<	3C
=	3D
>	3E
?	3F
@	40
A	41
B	42
C	43
D	44
E	45
F	46
G	47
H	48
I	49
J	4A
K	4B
L	4C
M	4D
N	4E
O	4F
P	50
Q	51
R	52
S	53
T	54
U	55
V	56
W	57
X	58
Y	59
Z	5A
[	5B
\	5C
]	5D
Λ (↑)	5E
— (↤)	5F
\	60
a	61
b	62
c	63
d	64
e	65
f	66
g	67
h	68
i	69
j	6A
k	6B
l	6C
m	6D
n	6E
o	6F
p	70
q	71
r	72
s	73
t	74
u	75
v	76
w	77
x	78
y	79
z	7A
{	7B
|	7C
} (ALT MODE)	7D
~	7E
DEL (RUB OUT)	7F

POWERS OF TWO

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 676 950 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125

POWERS OF 16 (IN BASE 10)

16^n	n	16^{-n}					
1	0	0.10000	00000	00000	00000	x	10
16	1	0.62500	00000	00000	00000	x	10^{-1}
256	2	0.39062	50000	00000	00000	x	10^{-2}
4 096	3	0.24414	06250	00000	00000	x	10^{-3}
65 536	4	0.15258	78906	25000	00000	x	10^{-4}
1 048 576	5	0.95367	43164	06250	00000	x	10^{-6}
16 777 216	6	0.59604	64477	53906	25000	x	10^{-7}
268 435 456	7	0.37252	90298	46191	40625	x	10^{-8}
4 294 967 296	8	0.23283	06436	53869	62891	x	10^{-9}
68 719 476 736	9	0.14551	91522	83668	51807	x	10^{-10}
1 099 511 627 776	10	0.90949	47017	72928	23792	x	10^{-12}
17 592 186 044 416	11	0.56843	41886	08080	14870	x	10^{-13}
281 474 976 710 656	12	0.35527	13678	80050	09294	x	10^{-14}
4 503 599 627 370 496	13	0.22204	46049	25031	30808	x	10^{-15}
72 057 594 037 927 936	14	0.13877	78780	78144	56755	x	10^{-16}
1 152 921 504 606 846 976	15	0.86736	17379	88403	54721	x	10^{-18}

POWERS OF 10 (IN BASE 16)

10^n	n	10^{-n}					
1	0	1.0000	0000	0000	0000		
A	1	0.1999	9999	9999	999A		
64	2	0.28F5	C28F	5C28	F5C3	x	16^{-1}
3E8	3	0.4189	374B	C6A7	EF9E	x	16^{-2}
2710	4	0.68DB	8BAC	710C	B296	x	16^{-3}
1 86A0	5	0.A7C5	AC47	1B47	8423	x	16^{-4}
F 4240	6	0.10C6	F7A0	B5ED	8D37	x	16^{-4}
98 9680	7	0.1AD7	F29A	BCAF	4858	x	16^{-5}
5F5 E100	8	0.2AF3	1DC4	6118	73BF	x	16^{-6}
3B9A CA00	9	0.44B8	2FA0	9B5A	52CC	x	16^{-7}
2 540B E400	10	0.6DF3	7F67	SEF6	EADF	x	16^{-8}
17 4876 E800	11	0.AFEB	FF0B	CB24	AAFF	x	16^{-9}
E8 D4A5 1000	12	0.1197	9981	2DEA	1119	x	16^{-9}
918 4E72 A000	13	0.1C25	C268	4976	81C2	x	16^{-10}
5AF3 107A 4000	14	0.2D09	370D	4257	3604	x	16^{-11}
3 8D7E A4C6 8000	15	0.480E	BE7B	9D58	566D	x	16^{-12}
23 8652 6FC1 0000	16	0.734A	CA5F	6226	F0AE	x	16^{-13}
163 4578 5D8A 0000	17	0.B877	AA32	36A4	B449	x	16^{-14}
DE0 B6B3 A764 0000	18	0.1272	5DD1	D243	ABA1	x	16^{-14}
8AC7 2304 89E8 0000	19	0.1D83	C94F	B6D2	AC35	x	16^{-15}

HEXADECIMAL-DECIMAL INTEGER CONVERSION

The table below provides for direct conversions between hexadecimal integers in the range 0-FFF and decimal integers in the range 0-4095. For conversion of larger integers, the table values may be added to the following figures:

Hexadecimal	Decimal	Hexadecimal	Decimal
01 000	4 096	20 000	131 072
02 000	8 192	30 000	196 608
03 000	12 288	40 000	262 144
04 000	16 384	50 000	327 680
05 000	20 480	60 000	393 216
06 000	24 576	70 000	458 752
07 000	28 672	80 000	524 288
08 000	32 768	90 000	589 824
09 000	36 864	A0 000	655 360
0A 000	40 960	B0 000	720 896
0B 000	45 056	C0 000	786 432
0C 000	49 152	D0 000	851 968
0D 000	53 248	E0 000	917 504
0E 000	57 344	F0 000	983 040
0F 000	61 440	100 000	1 048 576
10 000	65 536	200 000	2 097 152
11 000	69 632	300 000	3 145 728
12 000	73 728	400 000	4 194 304
13 000	77 824	500 000	5 242 880
14 000	81 920	600 000	6 291 456
15 000	86 016	700 000	7 340 032
16 000	90 112	800 000	8 388 608
17 000	94 208	900 000	9 437 184
18 000	98 304	A00 000	10 485 760
19 000	102 400	B00 000	11 534 336
1A 000	106 496	C00 000	12 582 912
1B 000	110 592	D00 000	13 631 488
1C 000	114 688	E00 000	14 680 064
1D 000	118 784	F00 000	15 728 640
1E 000	122 880	1 000 000	16 777 216
1F 000	126 976	2 000 000	33 554 432

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
010	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031
020	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047
030	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063
040	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079
050	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095
060	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111
070	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127
080	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143
090	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159
0A0	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175
0B0	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191
0C0	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207
0D0	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
0E0	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239
0F0	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
100	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271
110	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287
120	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303
130	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319
140	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0331	0333	0334	0335
150	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351
160	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367
170	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383
180	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399
190	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415
1A0	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431
1B0	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447
1C0	0448	0449	0450	0451	0452	0453	0454	0455	0456	0457	0458	0459	0460	0461	0462	0463
1D0	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	0479
1E0	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	0495
1F0	0496	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511
200	0512	0513	0514	0515	0516	0517	0518	0519	0520	0521	0522	0523	0524	0525	0526	0527
210	0528	0529	0530	0531	0532	0533	0534	0535	0536	0537	0538	0539	0540	0541	0542	0543
220	0544	0545	0546	0547	0548	0549	0550	0551	0552	0553	0554	0555	0556	0557	0558	0559
230	0560	0561	0562	0563	0564	0565	0566	0567	0568	0569	0570	0571	0572	0573	0574	0575
240	0576	0577	0578	0579	0580	0581	0582	0583	0584	0585	0586	0587	0588	0589	0590	0591
250	0592	0593	0594	0595	0596	0597	0598	0599	0600	0601	0602	0603	0604	0605	0606	0607
260	0608	0609	0610	0611	0612	0613	0614	0615	0616	0617	0618	0619	0620	0621	0622	0623
270	0624	0625	0626	0627	0628	0629	0630	0631	0632	0633	0634	0635	0636	0637	0638	0639
280	0640	0641	0642	0643	0644	0645	0646	0647	0648	0649	0650	0651	0652	0653	0654	0655
290	0656	0657	0658	0659	0660	0661	0662	0663	0664	0665	0666	0667	0668	0669	0670	0671
2A0	0672	0673	0674	0675	0676	0677	0678	0679	0680	0681	0682	0683	0684	0685	0686	0687
2B0	0688	0689	0690	0691	0692	0693	0694	0695	0696	0697	0698	0699	0700	0701	0702	0703
2C0	0704	0705	0706	0707	0708	0709	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719
2D0	0720	0721	0722	0723	0724	0725	0726	0727	0728	0729	0730	0731	0732	0733	0734	0735
2E0	0736	0737	0738	0739	0740	0741	0742	0743	0744	0745	0746	0747	0748	0749	0750	0751
2F0	0752	0753	0754	0755	0756	0757	0758	0759	0760	0761	0762	0763	0764	0765	0766	0767
300	0768	0769	0770	0771	0772	0773	0774	0775	0776	0777	0778	0779	0780	0781	0782	0783
310	0784	0785	0786	0787	0788	0789	0790	0791	0792	0793	0794	0795	0796	0797	0798	0799
320	0800	0301	0802	0803	0804	0805	0806	0807	0808	0809	0810	0811	0812	0813	0814	0815
330	0816	0817	0818	0819	0820	0821	0822	0823	0824	0825	0826	0827	0828	0829	0830	0831
340	0832	0833	0834	0835	0836	0837	0838	0839	0840	0841	0842	0843	0844	0845	0846	0847
350	0848	0849	0850	0851	0852	0853	0854	0855	0856	0857	0858	0859	0860	0861	0862	0863
360	0864	0865	0866	0867	0868	0869	0870	0871	0872	0873	0874	0875	0876	0877	0878	0879
370	0880	0881	0882	0883	0884	0885	0886	0887	0888	0889	0890	0891	0892	0893	0894	0895
380	0896	0897	0898	0899	0900	0901	0902	0903	0904	0905	0906	0907	0908	0909	0910	0911
390	0212	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923	0924	0925	0926	0927
3A0	0928	0929	0930	0931	0932	0933	0934	0935	0936	0937	0938	0939	0940	0941	0942	0943
3B0	0944	0945	0946	0947	0948	0949	0950	0951	0952	0953	0954	0955	0956	0957	0958	0959
3C0	0960	0961	0962	0963	0964	0965	0966	0967	0968	0969	0970	0971	0972	0973	0974	0975
3D0	0976	0977	0978	0979	0980	0981	0982	0983	0984	0985	0986	0987	0988	0989	0990	0991
3E0	0992	0993	0994	0995	0996	0997	0998	0999	1000	1001	1002	1003	1004	1005	1006	1007
3F0	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
400	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
410	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
420	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
430	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
440	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
450	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119
460	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135
470	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151
480	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167
490	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183
4A0	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199
4B0	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215
4C0	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231
4D0	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
4E0	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263
4F0	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279
500	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295
510	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311
520	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327
530	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343
540	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359
550	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375
560	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391
570	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407
580	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423
590	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439
5A0	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455
5B0	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471
5C0	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487
5D0	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503
5E0	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519
5F0	1520	1521	1522	1523	1524	1525	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535
600	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551
610	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
620	1568	1569	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
630	1584	1585	1586	1587	1588	1589	1590	1591	1592	1593	1594	1595	1596	1597	1598	1599
640	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	1613	1614	1615
650	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
660	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
670	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
680	1664	1665	1666	1667	1668	1669	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
690	1680	1681	1682	1683	1684	1685	1686	1687	1688	1689	1690	1691	1692	1693	1694	1695
6A0	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
6B0	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	1723	1724	1725	1726	1727
6C0	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
6D0	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
6E0	1760	1761	1762	1763	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
6F0	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
700	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	1802	1803	1804	1805	1806	1807
710	1808	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	1821	1822	1823
720	1824	1825	1826	1827	1828	1829	1830	1831	1832	1833	1834	1835	1836	1837	1838	1839
730	1840	1841	1842	1843	1844	1845	1846	1847	1848	1849	1850	1851	1852	1853	1854	1855
740	1856	1857	1858	1859	1860	1861	1862	1863	1864	1865	1866	1867	1868	1869	1870	1871
750	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
760	1888	1889	1890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1901	1902	1903
770	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919
780	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
790	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
7A0	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1967
7B0	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
7C0	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
7D0	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
7E0	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
7F0	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047
800	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
810	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
820	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
830	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
840	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
850	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
860	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
870	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
880	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
890	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8A0	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8B0	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8C0	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8D0	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8E0	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8F0	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303
900	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
910	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
920	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
930	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367
940	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
950	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399
960	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
970	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
980	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
990	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463
9A0	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479
9B0	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9C0	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9D0	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9E0	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9F0	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A00	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
A10	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591
A20	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A30	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	2618	2619	2620	2621	2622	2623
A40	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A50	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655
A60	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671
A70	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A80	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703
A90	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719
AA0	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
AB0	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
AC0	2752	2753	2754	2755	2756	2757	2758	2759	2760	4761	2762	2763	2764	2765	2766	2767
AD0	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AE0	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AF0	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815
B00	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831
B10	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
B20	2848	2849	2850	3851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863
B30	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2376	2877	2878	2879
B40	2880	2881	2882	2883	2884	2885	2866	2887	2888	2889	2890	2891	2892	2893	2894	2895
B50	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B60	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B70	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943
B80	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B90	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BA0	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991
BB0	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BC0	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BD0	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BE0	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BF0	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071
C00	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C10	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C20	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C30	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C40	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151
C50	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C60	3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C70	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C80	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	3210	3211	3212	3213	3214	3215
C90	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CA0	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CB0	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
CC0	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CD0	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CE0	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CF0	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327

HEXADECIMAL-DECIMAL INTEGER CONVERSION (Cont'd)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
D00	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D10	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D20	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370	3371	3372	3373	3374	3375
D30	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391
D40	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D50	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D60	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D70	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D80	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D90	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DA0	3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DB0	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
DC0	3520	3521	3522	3523	3524	3525	3526	3527	3528	3529	3530	3531	3532	3533	3534	3535
DD0	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DE0	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
DF0	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583
E00	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E10	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	3610	3611	3612	3613	3614	3615
E20	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E30	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E40	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E50	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E60	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E70	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E80	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E90	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EA0	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3758	3759
EB0	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775
EC0	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
ED0	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EE0	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EF0	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839
F00	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F10	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F20	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F30	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F40	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F50	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F60	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F70	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F80	3968	3969	3970	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F90	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FA0	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011	4012	4013	4014	4015
FB0	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	4028	4029	4030	4031
FC0	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FD0	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FE0	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FF0	4080	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095

F. ERROR MESSAGES

ERROR DETECTION AND REPORTING

The assemblers detect and report three classes of errors: source-file errors (including control line errors), run-time errors, and assembler control errors.

Source-file errors are indicated in the assembly listing by single-letter codes listed in column 1 of the erroneous source statement. If multiple errors occur in the same statement, only the first detected is reported. Each error is followed by a pointer to the previous erroneous line to ease finding errors. A summary of source-file errors is sent to the console and list devices.

Run-time errors cause the assembly to terminate abnormally. An error message of the form

```
error type      ERROR
```

is sent to the console and list device (if listing is in progress). The MONITOR assembler passes control to the Intellec monitor. The ISIS-II assembler returns control toISIS-II.

Assembler control errors in the assembler command are reported on the console device with the message

```
error type      ERROR
```

ISIS-II errors are shown as numerical codes. These are listed at the end of this appendix and explained in more detail in the *ISIS-II System User's Guide.*

ERROR CODES

Source-File Errors

Code	*Source*
B	Balance error. Parentheses or quote marks are unbalanced.
C	Control line error. An illegal control has been specified in a control line or a primary control appears in illegal context. The erroneous control and following controls on the same line are ignored.
D	Displacement error. In-page jump target address is out of range.

Code	*Source*
E	Expression error. An expression has been constructed erroneously; usually a missing operator or delimiter. Also caused by adjacent operands with no separating operator.
I	Illegal character. A statement contains an invalid ASCII character, or a specified number is illegal in the context of the number base in which it occurs. Also issued if a carriage return character is not followed by a line-feed character.
L	Location counter error. The symbol being defined has been illegally forward referenced. The definition is made in all cases except macro definitions. This condition can be corrected by moving the definition to precede all references.
M	Multiple definition. A symbol is illegally defined because of prior permanent definition. Only symbols defined by SET and MACRO are redefinable. All occurrences of the multiply-defined item are flagged.
N	Nesting error. Conditional assembly statements or macro body delimiters are improperly nested.
O	Opcode or operand illegal. An opcode or operand illegal in this particular device's instruction set causes a warning.
P	Phase error. Value of symbol being defined has changed between passes 1 and 2 of assembly. Caused by a forward reference of an operand in an ORG, IF, or DS directive. Also issued if source paper tape is changed between passes.
Q	Questionable syntax. Invalid syntax, usually due to a missing opcode.
R	Range error. The location counter exceeds the maximum memory of this processor.
U	Undefined symbol. Symbol used has not been defined.
V	Value illegal. Value exceeds permissible range for this operation or is null.
X	Illegal operand. Specified operand is illegal for this operation. Possible use of register-type symbol in illegal field or use of nonregister type in a field requiring register type.

Run-Time Errors

Message — *Explanation*

EOF ERROR

(ISIS-II assembler) End-of-file has been encountered before END directive or END was not terminated by a carriage-return, line-feed.

FILE ERROR

An ISIS file name used in an assembly-time command or control line is illegal or missing. Following this message, ISIS-II will report its own error number (see below).

MEMORY ERROR

System has insufficient memory to execute macro assembly. If MACROFILE is set, system must have at least 48K bytes of memory.

STACK ERROR

Assembler internal stack has overflowed. Possible causes of error:

1. Operators nested more than 16 deep;
2. More than 8 operands in DB or DW list;
3. More than 128 characters in an operand field (probably string too long);
4. Macro call or conditional assembly nesting greater than 8 deep;
5. INCLUDEs nested more than 4 deep.

TABLE ERROR

Assembler symbol or macro table has overflowed. More memory needed, or reduce number of symbols or macro definitions/calls.

Assembler Control Errors

PASS ERROR

Pass number specified is illegal or 'P=1' was not specified first. Entry is ignored.

COMMAND ERROR

Assembler control syntax is illegal, usually due to missing or illegal delimiter or missing parameter. The entire console command line is ignored.

ISIS-II Error Messages

By convention, error numbers 1-99 are reserved for errors that originate in or are detected by the resident routines of ISIS; error numbers 101-199 are reserved for user programs; numbers 200-255 are used for errors encountered by nonresident system routines. In the following list an asterisk precedes fatal errors. The other errors are generally nonfatal unless they are issued by the CONSOLE system call.

0 No error detected.
*1 Insufficient space in buffer area for a required buffer.
2 AFTN does not specify an open file.
3 Attempt to open more than six files simultaneously.
4 Illegal filename specification.
5 Illegal or unrecognized device specification in filename.
6 Attempt to write to a file open for input.
*7 Operation aborted; insufficient diskette space.
8 Attempt to read from a file open for output.
*9 No more room in diskette directory.
10 Filenames do not specify the same diskette.
11 Cannot rename file; name already in use.
12 Attempt to open a file already open.
13 No such file.
14 Attempt to open for writing (output or update) or to delete or rename a write-protected file.
*15 Attempt to load into ISIS area or buffer area.
*16 Incorrect ISIS binary format.
17 Attempt to rename or delete a file not on diskette.
18 Unrecognized system call.
19 Attempt to seek in a file not on diskette.
20 Attempt to seek backward past beginning of file.
21 Attempt to rescan a file not line edited.
22 Illegal ACCESS parameter to OPEN or access mode impossible for file specified (input mode for :LP:, for example).
23 No filename specified for a diskette file.
*24 Input/output error on diskette (see below).
25 Incorrect specification of echo file to OPEN.
26 Incorrect SWID parameter in ATTRIB system call.
27 Incorrect MODE parameter in SEEK system call.
28 Null file extension.
*29 End of file on console input.
*30 Drive not ready.
31 Attempted seek on file open for output.
32 Can't delete an open file.
33 Illegal system call parameter.
34 Bad RETSW parameter to LOAD.
35 Attempt to extend a file opened for input by seeking past end-of-file.

200	Unrecognized command.
201	Unrecognized switch.
202	Unrecognized delimiter character.
203	Invalid command syntax.
204	Premature end-of-file on input to HEXBIN.
205	Command line too long.
206	Illegal diskette label in FORMAT command.
207	No END statement in assembly language source code.
208	Checksum error in hexadecimal load format.

When error number 24 occurs, an additional message is output to the console

```
FDCC =00nn
```

where nn has the following meanings:

01	Deleted record.
02	CRC error (data field).
03	Invalid address mark.
04	Seek error.
08	Address error.
0A	CRC error (ID field).
0E	No address mark.
0F	Incorrect data address mark.
10	Data overrun or data underrun.
20	Write protect.
40	Write error.
80	Not ready.

INDEX

ADD Instructions
- A,#data *3-27*,A-4
- A,Rr *3-26*,A-4
- A,@Rr *3-27*,A-4

ADDC Instructions
- A,#data *3-28*,A-4
- A,Rr *3-26*,A-4
- A,@Rr *3-27*,A-4

ANL Instructions
- A,#data *3-17*,A-4
- A,Rr *3-14*,A-4
- A,@Rr *3-16*,A-4
- BUS,#data *3-18*,A-4
- Pp,#data *3-18*,A-4

ANLD Instruction *3-19*,A-4

Arithmetic Operations *2-3ff*,*3-23ff*
- Signed 2-4
- Two's Complement 2-5

Assembler
- Concepts 2-1ff,*6-1ff*
- Controls 2-1,*8-1ff*,C-1ff
- Directives 2-1,*5-1ff*,B-1ff
- Operation *9-1ff*
- Termination 5-7
- Versions 7-1

Auxiliary Carry Bit (AC) *1-7*, 1-9, 3-29

CALL Instruction *3-38*,A-5

Carry Bit (C) *1-7*,1-9,*3-29*

Clock 1-2,*3-34*

CLR Instructions
- CLR A *3-28*,A-5
- CLR C *3-29*,A-5
- CLR F0 *3-30*,A-5
- CLR F1 *3-30*,A-5

COND Control *8-5*,*8-8*,C-1

Conditional Assembly *5-5*

CPL Instructions
- CPL A *3-28*,A-5
- CPL C *3-30*,A-5
- CPL F0 *3-30*,A-5
- CPL F1 *3-31*,A-5

DA Instruction *3-29*,A-5

Data Bus Buffer (DBB) 1-12,*4-2ff*

Data Definition *5-3*

Data Memory 1-1,*1-3*

    Addressing 1-4

Data Transfer *3-2ff*

    Input/Output 3-11

    Within 8048 Memory 3-2

DB Directive *5-3*,B-2

DEBUG Control *8-3*,*8-7*,C-1

DEC Instructions

    DEC A *3-24*,A-5

    DEC Rr *3-24*,A-5

Designate Bank Flag (DBF) 3-35

DIS Instructions

    DIS I *3-31*,A-5

    DIS TCNTI *3-32*,A-5

DJNZ Instruction *3-25*,A-6

DS Directive *5-4*,B-2

DW Directive *5-4*,B-2

EJECT Control *8-5*,*8-9*,C-1

ELSE Directive *5-5*,B-2

EN Instructions

    EN I *3-31*,A-6

    EN TCNTI *3-32*,A-6

END Directive *5-7*,B-2

ENDIF Directive *5-5*,B-2

ENDM Directive *6-5*,B-3

ENTO Instruction *3-34*,A-6

EOT Directive 5-7,B-2

EQU Directive 5-3,B-2

Error Codes *F-1ff*

Event Counter 1-2,1-7

    Instructions *3-7*,*3-32*

EXITM Directive 6-*8*,B-3

Expander Device (8243) 1-2,*1-9*

Expressions 2-5

Flag Bit 0 (F0) *1-7*,1-8,*3-29ff*

Flag Bit 1 (F1) *1-7*,*3-29ff*

GEN Control *8-5*,C-3

- IF Directive *5-5*,B-2
- INC Instructions
  - INC A *3-23*,A-6
  - INC Rr *3-23*,A-6
  - INC @Rr *3-24*,A-6
- INCLUDE Control *8-4*,C-3
- IN Instruction
  - IN A,DBB *4-2*,A-6
  - IN A,Pp *3-11*,A-6
- Input/Output 1-10,*3-11*
  - Files 2-1,*7-2*
  - Port 1-2,1-10,3-11
  - Logical Operations *3-18*
- INS Instruction *3-12*,A-6
- Instruction Format 2-1ff
- Intellec MONITOR Assembler
  - Controls 8-7
  - Environment 7-1
  - Operation 9-4
- Interrupt Input Pin 1-8
- Interrupts 1-2,*1-8*
  - Controls *3-31*
- Interval Timer 1-2,*1-7ff*
  - Instructions *3-7,3-32*
- IRP Directive *6-6*,B-3
- IRPC Directive *6-7*,B-3
- ISIS-II Assembler
  - Controls 8-2
  - Environment 7-2
  - Operation 9-1

- Jump Instructions
  - JBb *3-45*,A-6
  - JC *3-41*,A-6
  - JF0 *3-42*,A-6
  - JF1 *3-42*,A-7
  - JMP *3-40*,A-7
  - JMPP *3-40*,A-7
  - JNC *3-41*,A-7
  - JN1 *3-44*,A-7
  - JNIBF *4-3*,A-7
  - JNT0 *3-43*,A-7
  - JNT1 *3-44*,A-7
  - JNZ *3-42*,A-7
  - JOBF *4-3*,A-7
  - JTF *3-44*,A-7
  - JT0 *3-43*,A-8
  - JT1 *3-43*,A-8
  - JZ *3-41*,A-8

- LIST Control *8-5,8-8,*C-1
- List File 2-1,*7-2,*D-1ff
- LOCAL Directive *6-5,*B-3
- Location Counter *5-2*
- Logical Operations 3-14

- MACRODEBUG Control *8-5,*C-3
- MACRO Directive *6-4,*B-3
- MACROFILE Control *8-4,*C-3
- Macros 2-1,*6-1ff*
- MCS-48
  - Component Family 1-1
  - Functional Overview 1-1ff
  - Instruction Set 3-1ff,A-1ff
- Memory Bank 1-1,*3-35*
- Memory Reservation 5-4
- MOD21 Control *8-4, 8-8,* C-1
- MOD41 Control *8-4,8-8,*C-1
- MONITOR Assembler (see 'Intellec')
- MOV Instructions
  - A,#data *3-5,*A-8
  - A,PSW *3-6,*A-8
  - A,Rr *3-2,*A-8
  - A,@Rr *3-3,*A-8
  - A,T *3-7,*A-8
  - PSW,A *3-6,*A-8
  - Rr,A *3-2,*A-8
  - Rr,#data *3-5,*A-8
  - @Rr,A *3-3,*A-8
  - @Rr,#data *3-5,*A-8
  - T,A *3-7,*A-9
- MOVD Instructions
  - A,Pp *3-13,*A-9
  - Pp,A *3-13,*A-9
- MOVP Instruction *3-8,*A-9
- MOVP3 Instruction *3-8,*A-9
- MOVX Instructions
  - A,@Rr *3-4,*A-9
  - @Rr,A *3-4,*A-9

- NOCOND Control *8-5,8-8,*C-1
- NODEBUG Control *8-3,8-7,*C-2
- NOGEN Control *8-5,*C-3
- NOLIST Control *8-5,8-8,*C-2
- NOMACRODEBUG Control *8-5*
- NOMACROFILE Control *8-4,*C-3
- NOOBJECT Control *8-3,*C-3
- NOPAGING Control *8-4,8-8,*C-2
- NOP Instruction *3-37,*A-9

- NOPRINT Control *8-3*,C-3
- NOSYMBOLS Control *8-3*,*8-7*,C-2
- NOXREF Control *8-4*,*8-8*,C-2
- Number Base Representation 2-4
- Number Range 2-4

- OBJECT Control *8-3*,C-3
- Object File 2-1,*7-2*
- Operators *2-6*,*6-9*,A-1
- ORG Directive *5-2*,B-2
- ORL Instructions
  - A,#data *3-17*,A-9
  - A,Rr *3-15*,A-9
  - A,@Rr *3-6*,A-9
  - BUS,#data *3-19*,A-9
  - Pp,#data *3-19*,A-10
- ORLD Instruction *3-20*,A-10
- Oscillator 1-2
- OUT Instruction *4-2*,A-10
- OUTL Instructions
  - BUS,A *3-12*,A-10
  - Pp,A *3-12*,A-10

- PAGELENGTH Control *8-4*,*8-8*,C-2
- PAGEWIDTH Control *8-4*,*8-8*,C-2
- PAGING Control *8-4*,*8-8*,C-2
- PRINT Control *8-3*,C-3
- Program Controls *1-6*,*3-29*
- Program Counter (PC) *1-2*
- Program Memory 1-1,*1-2*
  - Pages 1-2
- Program Status Word (PSW) 1-2,*1-9*,*3-6*

- Reference Tables E-1ff
- Register Bank
  - Selection *1-5*,1-8,*3-36*
  - Switch *1-8*,1-9
- REPT Directive *6-6*,B-3
- Reserved Words *2-10*,A-2
- RESTORE Control *8-5*,C-3
- RET Instruction *3-39*,A-10
- RETR Instruction *3-39*,A-10
- Rotate Instructions
  - RL *3-21*,A-10
  - RLC *3-22*,A-10
  - RR *3-21*,A-10
  - RRC *3-22*,A-10

- Sample Macros 6-14
- Sample Programs 3-45
- SAVE Control *8-5*,C-3
- SEL Instructions
  - MB0 *3-35*,A-11
  - MB1 *3-36*,A-11
  - RB0 *3-36*,A-11
  - RB1 *3-36*,A-11
- SET Directive *5-3*,B-2
- Single-Device Microcomputer 1-1
- Source File 2-1,*7-2*
- Source Program 2-1
- Stack *1-3*,1-5
- Stack Pointer *1-5*,1-8
- Status Register *1-12*,4-2
- STOP Instruction *3-33*,A-11
- STRT Instructions
  - STRT CNT *3-33*,A-11
  - STRT T *3-32*,A-11
- Subroutine Call 3-38
- SWAP Instruction *3-10*,A-11
- Symbol Cross-Reference File 2-1,*7-3*,D-5
- Symbols *2-9*
  - Characteristics 2-10ff
  - Definition 5-2
  - Reserved Symbols 2-10,A-2
- SYMBOLS Control *8-3*,*8-7*,C-2
- Symbol Table *2-9*,D-3

- Test Input 0 (T0) *1-8*,*3-34*
- Test Input 1 (T1) *1-8*
- Timer Flag *1-8*
- TITLE Control *8-4*,*8-9*,C-2

- UPI-41
  - Component Family 1-1,*1-11*
  - Hardware Overview 1-12
  - Instruction Set 4-1ff

- Working Registers 1-2,*1-3*
- XCH Instructions
  - A,Rr 3-9,A-11
  - A,@Rr 3-9,A-11

XCHD Instruction *3-10,*A-11
XREF Control *8-3,8-7,*C-2
XRL Instructions
A,#data *3-18,*A-11
A,Rr *3-15,*A-11
A,@Rr *3-16,*A-11

intel®

MCS-48 AND UPI-41
ASSEMBLY LANGUAGE MANUAL
98-255C

REQUEST FOR READER'S COMMENTS

The Microcomputer Division Technical Publications Department attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. ______

NAME ______ DATE ______
TITLE ______
COMPANY NAME/DEPARTMENT ______
ADDRESS ______
CITY ______ STATE ______ ZIP CODE ______

Please check here if you require a written reply. □

WE'D LIKE YOUR COMMENTS . . .

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

First Class
Permit No. 1040
Santa Clara, CA

BUSINESS REPLY MAIL

No Postage Stamp Necessary if Mailed in U.S.A.

Postage will be paid by

Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

Attention: MCD Technical Publications intel®

SOFTWARE PROBLEM REPORT

SUBMITTED BY:

Name ______________________________

Company ______________________________

Address ______________________________

____

Phone ____________________ Date ______________

FOR INTERNAL USE ONLY

No.

Date

Notes

Fix Date

Vers/System

CHECK ONE ITEM IN EACH CATEGORY

Product

- □ Software
- □ Manual

Product Type

- □ Monitor
- □ Assembler
- □ Compiler
- □ Simulator
- □ Editor
- □ Utility
- □ __________

Machine Line

- □ 4004/4040
- □ 8008
- □ 8080
- □ 3000
- □ __________

System

- □ Intellec
- □ Timeshare Co.

____

- □ In-House Computer

____

Exact Product/Manual Name ______________________________

Version Number (If not known, give date of receipt) ______________________________

PROBLEM:

REPLY:

PROBLEM DOCUMENTATION ATTACHED IS:

- □ Output Listing
- □ Program Listing
- □ Paper Tape Program Source
- □ __________

WE'D LIKE YOUR COMMENTS . . .

This document is one of a series describing Intel software products. Your comments on the back of this form will help us produce better software and manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

First Class
Permit No. 1040
Santa Clara, CA

BUSINESS REPLY MAIL

No Postage Stamp Necessary if Mailed in U.S.A.

Postage will be paid by:

Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

Attention: MCS Systems Marketing